Virtual University of Pakistan
FEDERAL GOVERNMENT UNIVERSITY

COURSE HANDOUTS

Introduction to Programming

CS201

Order Your VU Books Now

Get all Virtual University books in high-quality hard copy, delivered right to your doorstep. Visit vubookshoppk.com

Order Now

Lecture No. 1 3
Lecture No. 2 9
Lecture No. 3 15
Lecture No. 4 24
Lecture No. 5 33
Lecture No. 6 45
Lecture No. 7 55
Lecture No. 8 66
Lecture No. 9 77
Lecture No. 10 88
Lecture No. 11 99
Lecture No. 12 112
Lecture No. 13 124
Lecture No. 14 143
Lecture No. 15 158
Lecture No. 16 ..... **171**
Lecture No. 17 186
Lecture No. 18 198
Lecture No. 19 211
Lecture No. 20 228
Lecture No. 21 245
Lecture No. 22 255
Lecture No. 23 268
Lecture No. 24 278
Lecture No. 25 291
Lecture No. 26 304
Lecture No. 27 318
Lecture No. 28 328
Lecture No. 29 344
Lecture No. 30 358
Lecture No. 31 368
Lecture No. 32 381
Lecture No. 33 394
Lecture No. 34 407
Lecture No. 35 421
Lecture No. 36 432
Lecture No. 37 443
Lecture No. 38 453
Lecture No. 39 468
Lecture No. 40 481
Lecture No. 41 496
Lecture No. 42 509
Lecture No. 43 519
Lecture No. 44 528
Lecture No. 45 552

vubookshoppk.com

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Lecture No. 1

Summary

- What is programming
- Why programming is important
- What skills are needed
- Develop a basic recipe for writing programs
- Points to remember

What is programming

As this course is titled “Introduction to programming”, therefore it is most essential and appropriate to understand what programming really means. Let us first see a widely known definition of programming.

Definition: *"A program is a precise sequence of steps to solve a particular problem.”*

It means that when we say that we have a program, it actually means that we know about a complete set activities to be performed in a particular order. The purpose of these activities is to solve a given problem.

Alan Perlis, a professor at Yale University, says:

"It goes against the grain of modern education to teach children to program. What fun is there in making plans, acquiring discipline in organizing thoughts, devoting attention to detail and learning to be self-critical? "

It is a sarcastic statement about modern education, and it means that the modern education is not developing critical skills like planning, organizing and paying attention to detail. Practically, in our day to day lives we are constantly planning, organizing and paying attention to fine details (if we want our plans to succeed). And it is also fun to do these activities. For example, for a picnic trip we plan where to go, what to wear, what to take for lunch, organize travel details and have a good time while doing so.

When we talk about computer programming then as Mr. Steve Summit puts it

“At its most basic level, programming a computer simply means telling it what to do, and this vapid-sounding definition is not even a joke. There are no other truly fundamental aspects of computer programming; everything else we talk about will simply be the details of a particular, usually artificial, mechanism for telling a computer what to do. Sometimes these mechanisms are chosen because they have been found to be convenient for programmers (people) to use; other times they have been chosen because they're easy for the computer to understand. The first hard thing about programming is to learn, become comfortable with, and accept these artificial mechanisms, whether they make ``sense'' to you or not. “

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Why Programming is important

The question most of the people ask is why should we learn to program when there are so many application software and code generators available to do the task for us. Well the answer is as give by the Matthias Felleisen in the book 'How to design programs'

"The answer consists of two parts. First, it is indeed true that *traditional forms of programming* are useful for just a few people. But, programming *as we the authors understand it* is useful for everyone: the administrative secretary who uses spreadsheets as well as the high-tech programmer. In other words, we have a broader notion of programming in mind than the traditional one. We explain our notion in a moment. Second, we teach our idea of programming with a technology that is based on the principle of minimal intrusion. Hence, our notion of programming teaches problem-analysis and problem-solving skills *without* imposing the overhead of traditional programming notations and tools."

Hence learning to program is important because it develops **analytical** and **problem solving** abilities. It is a creative activity and provides us a mean to express abstract ideas. Thus programming is fun and is much more than a vocational skill. By designing programs, we learn many skills that are important for all professions. These skills can be summarized as:

- Critical reading
- Analytical thinking
- Creative synthesis

vubookshoppk.com

What skills are needed

Programming is an important activity as people life and living depends on the programs one make. Hence while programming one should

- Paying attention to detail
- Think about the reusability.
- Think about user interface
- Understand the fact the computers are stupid
- Comment the code liberally

Paying attention to detail

In programming, the details matter. This is a very important skill. A good programmer always analyzes the problem statement very carefully and in detail. You should pay attention to all the aspects of the problem. You can't be vague. You can't describe your program 3/4th of the way, then say, "You know what I mean?", and have the compiler figure out the rest.

Furthermore you should pay attention to the calculations involved in the program, its flow, and most importantly, the logic of the program. Sometimes, a grammatically correct sentence does not make any sense. For example, here is a verse from poem "Through the Looking Glass" written by Lewis Carol:

"Twas brillig, and the slithy toves
Did gyre and gimble in the wabe "

The grammar is correct but there is no meaning. Similarly, the sentence, "Mr. ABC sleeps thirty hours every day", is grammatically correct but it is illogical.

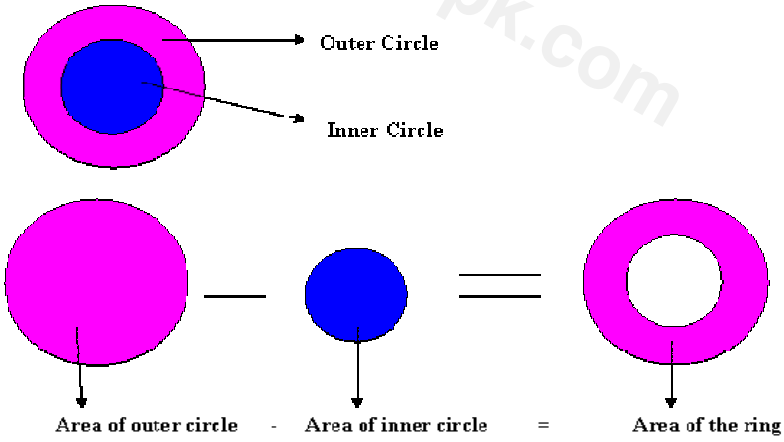
So it may happen that a program is grammatically correct. It compiles and runs but produces incorrect or absurd results and does not solve the problem. It is very important to pay attention to the logic of the program.

Think about the reusability

When ever you are writing a program, always keep in mind that it could be reused at some other time. Also, try to write in a way that it can be used to solve some other related problem. A classic example of this is:

Suppose we have to calculate the area of a given circle. We know the area of a circle is (Pi * r^2). Now we have written a program which calculates the area of a circle with given radius. At some later time we are given a problem to find out the area of a ring. The area of the ring can be calculated by subtracting the area of outer circle from the area of the inner circle. Hence we can use the program that calculates the area of a circle to calculate the area of the ring.

vubookshoppk.com



Think about Good user interface

As programmers, we assume that computer users know a lot of things, this is a big mistake. So never assume that the user of your program is computer literate. Always provide an easy to understand and easy to use interface that is self explanatory.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Understand the fact that computers are stupid

Computers are incredibly stupid. They do *exactly* what you tell them to do: no more, no less-- unlike human beings. Computers can't think by themselves. In this sense, they differ from human beings. For example, if someone asks you, “What is the time?”, “Time please?” or just, “Time?” you understand anyway that he is asking the time but computer is different. Instructions to the computer should be explicitly stated. Computer will tell you the time only if you ask it in the way you have programmed it.

When you're programming, it helps to be able to "think" as stupidly as the computer does, so that you are in the right frame of mind for specifying everything in minute detail, and not assuming that the right thing will happen by itself.

Comment the code liberally

Always comment the code liberally. The comment statements do not affect the performance of the program as these are ignored by the compiler and do not take any memory in the computer. Comments are used to explain the functioning of the programs. It helps the other programmers as well as the creator of the program to understand the code.

vubookshoppk.com

Program design recipe

In order to design a program effectively and properly we must have a recipe to follow. In the book name ‘How to design programs’ by Matthias Felleisen.and the co-worker, the idea of design recipe has been stated very elegenlty as

“Learning to design programs is like learning to play soccer. A player must learn to trap a ball, to dribble with a ball, to pass, and to shoot a ball. Once the player knows those basic skills, the next goals are to learn to play a position, to play certain strategies, to choose among feasible strategies, and, on occasion, to create variations of a strategy because none fits. “

The author then continue to say that:
“A programmer is also very much like an architect, a composers, or a writer. They are creative people who start with ideas in their heads and blank pieces of paper. They conceive of an idea, form a mental outline, and refine it on paper until their writings reflect their mental image as much as possible. As they bring their ideas to paper, they employ basic drawing, writing, and playing music to express certain style elements of a building, to describe a person's character, or to formulate portions of a melody. They can practice their trade because they have honed their basic skills for a long time and can use them on an instinctive level.
Programmers also form outlines, translate them into first designs, and iteratively refine them until they truly match the initial idea. Indeed, the best programmers edit and rewrite their programs many times until they meet certain aesthetic standards. And just like soccer players, architects, composers, or writers, programmers must practice the basic skills of their trade for a long time before they can be truly creative.
Design recipes are the equivalent of soccer ball handling techniques, writing techniques, arrangements, and drawing skills. “

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Hence to design a program properly, we must:

- Analyze a problem statement, typically expressed as a word problem.
- Express its essence, abstractly and with examples.
- Formulate statements and comments in a precise language.
- Evaluate and revise the activities in light of checks and tests and
- Pay attention to detail.

All of these are activities that are useful, not only for a programmer but also for a businessman, a lawyer, a journalist, a scientist, an engineer, and many others.

Let us take an example to demonstrate the use of design recipe:

Suppose we have to develop a payroll system of a company. The company has permanent staff, contractual staff, hourly based employees and per unit making employees. Moreover, there are different deductions and benefits for permanent employees and there is a bonus for per unit making employees and overtime for contractual employees.

We need to analyze the above problem statement. The company has four categories of employees; i.e.; Permanent staff, Contractual staff, hourly based employees and per unit making employees. Further, permanent staff has benefits and deductions depending upon their designation. Bonus will be given to per unit making employees if they make more than 10 pieces a day. Contractual employee will get overtime if they stay after office hours.

Now divide the problem into small segments and calculations. Also include examples in all segments. In this problem, we should take an employee with his details from each category. Let's say, Mr. Ahmad is a permanent employee working as Finance Manager. His salary is Rs.20000 and benefits of medical, car allowance and house rent are Rs.4000 and there is a deduction of Rs.1200. Similarly, we should consider employees from other categories. This will help us in checking and testing the program later on.

The next step is to formulate these statements in a precise language, i.e. we can use the pseudo code and flowcharting. which will be then used to develop the program using computer language.

Then the program should be evaluated by testing and checking. If there are some changes identified, we revise the activities and repeat the process. Thus repeating the cycle, we achieve a refined solution.

Points to remember

Hence the major points to keep in mind are:

- Don't assume on the part of the users

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

- User Interface should be friendly
- Don't forget to comment the code
- PAY ATTENTION TO DETAIL
- Program, program and program, not just writing code, but the whole process of design and development

vubookshoppk.com

Lecture No. 2

Reading Material

Deitel & Deitel – C++ How to Program	chapter 1 1.2, 1.3, 1.4, 1.6, 1.7 1.11, 1.12, 1.13

Summary

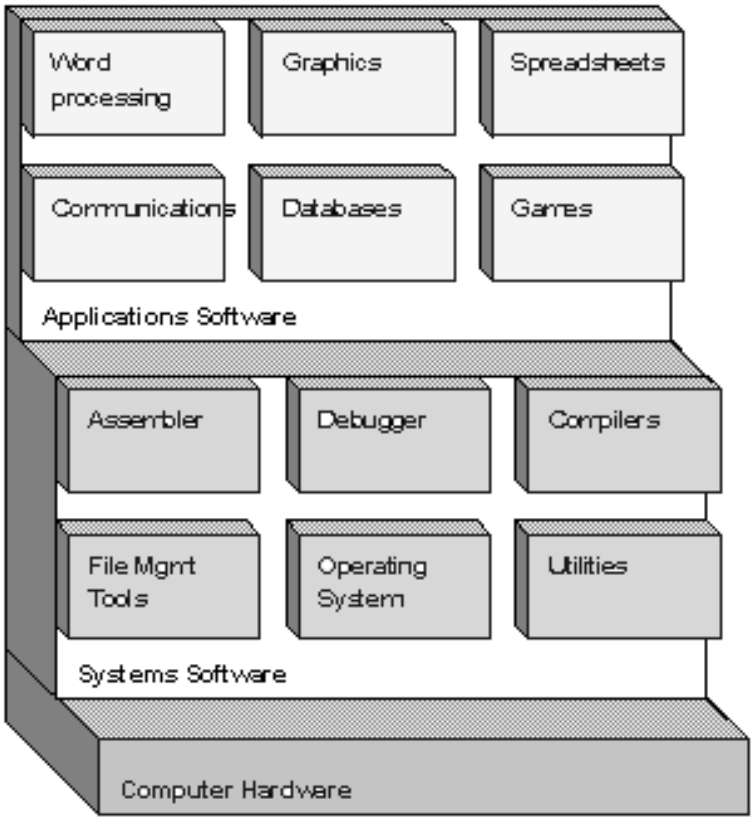
- Software Categories
  - System Software
  - Application Software
- History of C language
- Development Environment of 'C'

Software Categories

Software is categorized into two main categories

- System Software
- Application Software

vubookshoppk.com



System Software

The system software controls the computer. It communicates with computer's hardware (key board, mouse, modem, sound card etc) and controls different aspects of operations. Sub categories of system software are:

- Operating system
- Device drivers
- Utilities

Operating system

An operating system (sometimes abbreviated as "OS") is the program that manages all the other programs in a computer. It is a integrated collection of routines that service the sequencing and processing of programs by a **computer**. *Note:* An operating **system** may provide many services, such as resource allocation, scheduling, **input/output** control, and **data management**.

Definition

"Operating system is the software responsible for controlling the allocation and usage of hardware resources such as memory, central processing unit (CPU) time, disk space, and peripheral devices. The operating system is the foundation on which applications, such as word processing and spreadsheet programs, are built. (*Microsoft*)"

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Device drivers

The device driver software is used to communicate between the devices and the computer. We have monitor, keyboard and mouse attached to almost all PC's; if we look at the properties of these devices we will see that the operating system has installed special software to control these devices. This piece of software is called device driver software. When we attach a new device with the computer, we need software to communicate with this device. These kinds of software are known as device drivers e.g. CD Rom driver, Sound Card driver and Modem driver. Normally manufacturer of the device provide the device driver software with the device. For scanners to work properly with the computers we install the device driver of the scanner. Nowadays if you have seen a scanner, it comes with TWAIN Drivers. TWAIN stands for Technology Without An Interesting Name.

Utility Software

Utility software is a program that performs a very specific task, usually related to managing system resources. You would have noticed a utility of Disk Compression. Whenever you write a file and save it to the disk, Compression Utility compresses the file (reduce the file size) and write it to the disk and when you request this file from the disk, the compression utility uncompressed the file and shows its contents. Similarly there is another utility, Disk Defragmentation which is used to defragment the disk. The data is stored on the disks in chunks, so if we are using several files and are making changes to these files then the different portions of file are saved on different locations on the disk. These chunks are linked and the operating system knows how to read the contents of file from the disk combining all the chunks. Similarly when we delete a file then the place where that file was stored on the disk is emptied and is available now to store other files. As the time goes on, we have a lot of empty and used pieces on the disk. In such situation we say that the disk is fragmented now. If we remove this fragmentation the chunks of data on the disk will be stored close to each other and thus reading of data will be faster. For the purpose of removing fragmentation on the disk the Defragmentation utility is used.

The compilers and interpreters also belong to the System Software category.

Application software

A program or group of programs designed for end users**.** For example a program for Accounting, Payroll, Inventory Control System, and guided system for planes. GPS (global positioning system), another application software, is being used in vehicles, which through satellite determines the geographical position of the vehicle

History of C language

The C language was developed in late 60's and early 70's, in Bell Laboratories. In those days BCPL and B languages were developed there. The BCPL language was developed in 1967 by Martin Richards as a language for writing operating systems software and compilers. In 1970 Ken Thompson used B language to create early

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

versions of the UNIX operating system at Bell Laboratories. Thus both the languages were being used to develop various system software even compilers. Both BCPL and B were 'type less' languages, every data item occupied one 'word' in memory and the burden of treating a data item as a whole number or real number, for example was the responsibility of the programmer.

Dennis Ritchie developed a general purpose language, called C language, by using different features of BCPL and B languages. C uses many important concepts of BCPL and B while adding data typing and other features. In the start C became widely known as the development language of the UNIX operating system, and the UNIX operating system was written by using this C language. The C language is so powerful that the compiler of C and other various operating systems are written in C. C language has almost unlimited powers to do with computers. You can program to turn on or off any device of computer. You can do a lot to hard disk and other peripherals. It is very easy to write a program in C that stops the running of computer. So be careful while programming in C.

The C language and UNIX operating system widely spread in educational and research institutions. There was C and UNIX everywhere. Due to the wide spread of C, different researchers started to add their features in the language. And thus different variations in C came into existence. Many universities developed their own C by adding different features to the C language developed by Ritchie. These variations led to the need of a standard version of C. In 1983 a technical committee was created under the American National Standards Committee on Computer and Information Processing to provide an unambiguous and machine-independent definition of the language. In 1989 the standard was approved. ANSI cooperated with the International Standard Organization (ISO) to standardize C worldwide.

Tools of the trade

As programmer we need different tools to develop a program. These tools are needed for the life cycle of programs

Editors

First of all we need a tool for writing the code of a program. For this purpose we used Editors in which we write our code. We can use word processor too for this, but word processors have many other features like bold the text, italic, coloring the text etc, so when we save a file written in a word processor, lot of other information including the text is saved on the disk. For programming purposes we don't need these things we only need simple text. Text editors are such editors which save only the text which we type. So for programming we will be using a text editor

Compiler and Interpreter

As we write the code in English and we know that computers can understand only 0s and 1s. So we need a translator which translates the code of our program into machine language. There are two kinds of translators which are known as Interpreter and Compilers. These translators translate our program which is written in C-Language into Machine language. Interpreters translates the program line by line meaning it reads one line of program and translates it, then it reads second line, translate it and so on. The benefit of it is that we get the errors as we go along and it is very easy to correct the errors. The drawback of the interpreter is that the program executes slowly

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

as the interpreter translates the program line by line. Another drawback is that as interpreters are reading the program line by line so they cannot get the overall picture of the program hence cannot optimize the program making it efficient.

Compilers also translate the English like language (Code written in C) into a language (Machine language) which computers can understand. The Compiler read the whole program and translates it into machine language completely. The difference between interpreter and compiler is that compiler will stop translating if it finds an error and there will be no executable code generated whereas Interpreter will execute all the lines before error and will stop at the line which contains the error. So Compiler needs syntactically correct program to produce an executable code. We will be using compiler in our course

Debugger

Another important tool is Debugger. Every programmer should be familiar with it. Debugger is used to debug the program i.e. to correct the logical errors. Using debugger we can control our program while it is running. We can stop the execution of our program at some point and can check the values in different variables, can change these values etc. In this way we can trace the logical errors in our program and can see whether our program is producing the correct results. This tool is very powerful, so it is complex too

Linker

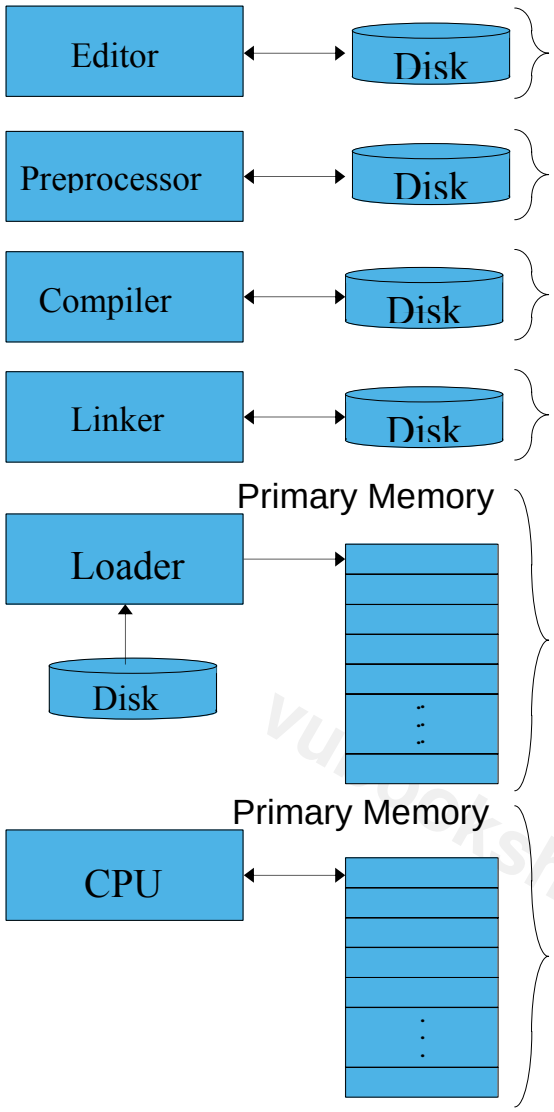
Most of the time our program is using different routines and functions that are located in different files, hence it needs the executable code of those routines/functions. Linker is a tool which performs this job, it checks our program and includes all those routines or functions which we are using in our program to make a standalone executable code and this process is called Linking

Loader

After a executable program is linked and saved on the disk and it is ready for execution. We need another process which loads the program into memory and then instruct the processor to start the execution of the program from the first instruction (the starting point of every C program is from the main function). This processor is known as loader. Linker and loaders are the part of development environment. These are part of system software.

The following figure represents a graphical explanation of all the steps involved in writing and executing a program.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533



Preprocessor program processes the code.

Compiler creates object code and stores it on disk.

Linker links the object code with the libraries

Loader puts program in memory.

CPU takes each instruction and executes it, possibly storing new data values as the program executes.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Lecture No. 3

Reading Material

Deitel & Deitel – C++ How to Program chapter 1
1.19, 1.20, 1.21,
1.22

Summary

First C program

Variables
Data Types
Arithmetic Operators
Precedence of Operators
Tips

vubookshoppk.com

First C program

The best way to learn C is to start coding right away. So here is our very first program in C.

```
# include <iostream.h>

main()
{
    cout << "Welcome to Virtual University of Pakistan";
}
```

We will look at this code line by line and try to understand them.

include <iostream.h>
#include: This is a pre-processor directive. It is not part of our program; it is an instruction to the compiler. It tells the C compiler to **include** the contents of a file, in this case the system file **iostream.h**. The compiler knows that it is a system file, and therefore looks for it in a special place. The features of preprocessor will be discussed later. For the time being take this line on faith. You have to write this line. The sign # is known as HASH and also called SHARP.

<iostream.h>
This is the name of the library definition file for all Input Output Streams. Your program will almost certainly want to send stuff to the screen and read things from the keyboard. iostream**.h** is the name of the file in which has code to do that work for you

main()
The name **main** is special, in that the **main** is actually the one which is run when your program is used. A C program is made up of a large number of functions. Each of these is given a name by the programmer and they refer to each other as the program runs. C regards the name "**main**" as a special case and will run this function first. If you forget to have a main function, or mistype the name, the compiler will give you an error.

Notice that there are parentheses ("**( )**", normal brackets) with main. Here the parentheses contain nothing. There may be something written inside the parentheses. It will be discussed in next lectures.
{ }
Next, there is a curly bracket also called braces("**{ }**"). For every open brace there must be a matching close. Braces allows to group together pieces of a program. The body of main is enclosed in braces. Braces are very important in C; they enclose the blocks of the program.
cout << " Welcome to Virtual University of Pakistan"
cout:
This is known as out put stream in C and C++. Stream is a complicated thing, you will learn about it later. Think a stream as a door. The data is transferred through stream, **cout** takes data from computer and sends it to the output. For the moment it is a screen of the monitor. hence we use **cout** for output.
<<
The sign **<<** indicates the direction of data. Here it is towards **cout** and the function of **cout** is to show data on the screen.
" Welcome to Virtual University of Pakistan"
The thing between the double quotes (" ") is known as character string. In C programming character strings are written in double quotes. Whatever is written after << and within quotation marks will be direct it to **cout,** cout wil**l** display it on the screen.
;
There is a semicolon (**;**) at the end of the above statement. This is very important. All C statements end with semicolon (**;**). Missing of a semicolon (**;**) at the end of statement is a syntax error and compiler will report an error during compilation. If there is only a semicolon (**;**) on a line than it will be called a null statement. i.e. it does nothing. The extra semicolons may be put at the end but are useless and aimless. Do not put semicolon (**;**) at a wrong place, it may cause a problem during the execution of the program or may cause a logical error.
In this program we give a fixed character string to **cout** and the program prints it to the screen as:
Variables
During programming we need to store data. This data is stored in variables. Variables are locations in memory for storing data. The memory is divided into blocks. It can be viewed as pigeon-holes. You can also think of it as PO Boxes. In post offices there are different boxes and each has an address. Similarly in memory, there is a numerical address for each location of memory (block). It is difficult for us to handle these numerical addresses in our programs. So we give a name to these locations. These names are variables. We call them variables because they can contain different values at different times.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

The variable names in C may be started with a character or an underscore (_). But avoid starting a name with underscore (_). C has many libraries which contain variables and function names normally starting with underscore (_). So your variable name starting with underscore (_) may conflict with these variables or function names.

In a program every variable has

Name

Type

Size

Value

The variables having a name, type and size (type and size will be discussed later) are just empty boxes. They are useless until we put some value in them. To put some value in these boxes is known as assigning values to variables. In C language, we use assignment operator for this purpose.

Assignment Operator

In C language equal-to-sign (=**)** is used as assignment operator. Do not confuse the algebraic equal-to with the assignment operator. In Algebra X = 2 means the value of X is 2, whereas in C language X = 2 (where X is a variable name) means take the value 2 and put it in the memory location labeled as X, afterwards you can assign some other value to X, for example you can write X = 10, that means now the memory location X contains the value 10 and the previous value 2 is no more there. Assignment operator is a binary operator (a binary operator has two operands). It must have variable on left hand side and expression (that evaluates to a single value) on right hand side. This operator takes the value on right hand side and stores it to the location labeled as the variable on left hand side, e.g. X = 5, X = 10 + 5, and X = X +1.

In C language the statement X = X + 1 means that add 1 to the value of X and then store the result in X variable. If the value of X is 10 then after the execution of this statement the value of X becomes 11. This is a common practice for incrementing the value of the variable by ‘one in C language. Similarly you can use the statement X = X - 1 for decrementing the value of the variable by one. The statement X = X + 1 in algebra is not valid except when X is infinity. So do not confuse assignment operator (=) with equal sign (=) in algebra. Remember that assignment operator must have a variable name on left hand side unlike algebra in which you can use expression on both sides of equal sign (=). For example, in algebra, X +5 = Y + 7 is correct but incorrect in C language. The compiler will not understand it and will give error.

Data Types

A variable must have a data type associated with it, for example it can have data types like integer, decimal numbers, characters etc. The variable of type Integer stores integer values and a character type variable stores character value. The primary difference between various data types is their size in memory. Different data types have different size in memory depending on the machine and compilers. These also affect the way they are displayed. The ‘**cout**’ knows how to display a digit and a character. There are few data types in C language. These data types are reserved words of C language. The reserve words can not be used as a variable name.

Let’s take a look into different data types that the C language provides us to deal with whole numbers, real numbers and character data.

Whole Numbers

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

The C language provides three data types to handle whole numbers.

int
short
long

int Data Type

The data type int is used to store whole numbers (integers). The integer type has a space of 4 bytes (32 bits for windows operating system) in memory. And it is mentioned as '**int**' which is a reserved word of C, so we can not use it as a variable name.

In programming before using any variable name we have to declare that variable with its data type. If we are using an integer variable named as '**i**', we have to declare it as

```
int i ;
```

The above line is known as declaration statement. When we declare a variable in this way, it reserves some space in memory depending on the size of data type and labels it with the variable name. The declaration statement **int i ;** reserves 4 bytes of memory and labels it as '**i**'. This happens at the execution time.

vubookshoppk.com

Sample Program 1

Let's consider a simple example to explain int data type. In this example we take two integers, add them and display the answer on the screen.
The code of the program is written below.

```
#include <iostream.h>
main()
{
        int x;
        int y;
        int z;
        x = 5;
        y = 10;
        z = x + y;
        cout << "x = ";
        cout << x;
        cout << " y=";
        cout << y;
        cout  << " z = x + y = ";
        cout << z;
}
```

The first three lines declare three variables x, y and z as following.

```
        int x;
        int y;
        int z;
```

These three declarations can also be written on one line. C provides us the comma separator (,). The above three lines can be written in a single line as below

```
int x, y, z;
```

As we know that semicolon (;) indicates the end of the statement. So we can write many statements on a single line. In this way we can also write the above declarations in the following form

```
int x; int y; int z;
```

For good programming practice, write a single statement on a single line.
Now we assign values to variables x and y by using assignment operator. The lines x = 5; and y = 10 assign the values 5 and 10 to the variables x and y, respectively. These statements put the values 5 and 10 to the memory locations labeled as x and y.
The next statement z = x + y; evaluates the expression on right hand side. It takes values stored in variables x and y (which are 5 and 10 respectively), adds them and by using the assignment operator (=), puts the value of the result, which is 15 in this case, to the memory location labeled as z.
Here a thing to be noted is that the values of x and y remains the same after this operation. In arithmetic operations the values of variables used in expression on the right hand side are not affected. They remain the same. But a statement like x = x + 1; is an exceptional case. In this case the value of x is changed.

The next line cout << " x = " ; is simple it just displays ' x = ' on the screen.
Now we want to display the value of x after 'x ='. For this we write the statement cout << x ;
Here comes the affect of data type on **cout**. The previous statement cout << "x = " ; has a character string after << sign and **cout** simply displays the string. In the statement cout << x; there is a variable name x. Now cout will not display 'x' but the value of x. The cout interprets that x is a variable of integer type, it goes to the location x in the memory and takes its value and displays it in integer form, on the screen. The next line cout << "y ="; displays ' y = ' on the screen. And line cout << y; displays the value of y on the screen. Thus we see that when we write something in quotation marks it is displayed as it is but when we use a variable name it displays the value of the variable not name of the variable. The next two lines cout << "z = x + y = "; and cout << z; are written to display 'z = x + y = ' and the value of z that is 15.
Now when we execute the program after compiling, we get the following output.

```
x = 5 y = 10 z = x + y = 15
```

short Data type

We noted that the integer occupies four bytes in memory. So if we have to store a small integer like 5, 10 or 20 four bytes would be used. The C provides another data type for storing small whole numbers which is called short. The size of short is two bytes and it can store numbers in range of -32768 to 32767. So if we are going to use a variable for which we know that it will not increase from 32767, for example the age of different people, then we use the data type short for age. We can write the above sample program by using short instead of int.

```
/*This program uses short data type to store values */
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
#include <iostream.h>
main()
{
        short x;
        short y;
        short z;
        x = 5;
        y = 10;
        z = x + y;
        cout << "x = ";
        cout << x;
        cout << " y=";
        cout << y;
        cout  << " z = x + y = ";
        cout << z;
}
```

long Data Type

On the other side if we have a very large whole number that can not be stored in an int then we use the data type long provided by C. So when we are going to deal with very big whole numbers in our program, we use long data type. We use it in program as:

long x = 300500200;

Real Numbers

The C language provides two data types to deal with real numbers (numbers with decimal points e.g. 1.35, 735.251). The real numbers are also known as floating point numbers.

float
double

float Data Type

To store real numbers, float data type is used. The float data type uses four bytes to store a real number. Here is program that uses float data types.

/*This program uses short data type to store values */

```
#include <iostream.h>
main()
{
        float x;
        float y;
        float z;
```

vubookshoppk.com

```
        x = 12.35;
        y = 25.57;
        z = x + y;
        cout << " x = ";
        cout << x;
        cout << " y = ";
        cout << y;
        cout << " z = x + y = ";
        cout << z;
}
```

double Data Type

If we need to store a large real number which cannot be store in four bytes, then we use **double** data type. Normally the size of double is twice the size of float. In program we use it as:

```
double x = 345624.769123;
```

char Data Type

So far we have been looking on data types to store numbers, In programming we do need to store characters like a,b,c etc. For storing the character data C language provides **char** data type. By using char data type we can store characters in variables. While assigning a character value to a char type variable single quotes are used around the character as 'a'.

```
/* This program uses short data type to store values */

#include <iostream.h>
main()
{
        char x;
        x = 'a';
        cout << "The character value in x =  ";
        cout << x;
}
```

Arithmetic Operators
In C language we have the usual arithmetic operators for addition, subtraction, multiplication and division. C also provides a special arithmetic operator which is called modulus. All these operators are binary operators which means they operate on two operands. So we need two values for addition, subtraction, multiplication, division and modulus.

ARITHMETIC OPERATION	ARITHMETIC OPERATOR	ALGEBRAIC EXPRESSION	C EXPRESSION
Addition	+	x + y	x + y
Subtraction	-	x – y	x - y

Multiplication	*	Xy	x * y
Division	/	x ÷ y, x / y	x / y
Modulus	%	x mod y	x % y

Addition, subtraction and multiplication are same as we use in algebra.
There is one thing to note in division that when we use integer division (i.e. both operands are integers) yields an integer result. This means that if, for example, you are dividing 5 by 2 (5 / 2) it will give integer result as 2 instead of actual result 2.5. Thus in integer division the result is truncated to the whole number, the fractional part (after decimal) is ignored. If we want to get the correct result, then we should use float data type.
The modulus operator returns the remainder after division. This operator can only be used with integer operands. The expression **x % y** returns the remainder after x is divided by y. For example, the result of 5 % 2 will be 1, 23 % 5 will be 3 and 107%10 will be 7.
Precedence of Operators
The arithmetic operators in an expression are evaluated according to their precedence. The precedence means which operator will be evaluated first and which will be evaluated after that and so on. In an expression, the parentheses ( ) are used to force the evaluation order. The operators in the parentheses ( ) are evaluated first. If there are nested parentheses then the inner most is evaluated first.
The expressions are always evaluated from left to right. The operators *, / and % have the highest precedence after parentheses. These operators are evaluated before + and – operators. Thus + and – operators has the lowest precedence. It means that if there are * and + operators in an expression then first the * will be evaluated and then its result will be added to other operand. If there are * and / operators in an expression (both have the same precedence) then the operator which occurs first from left will be evaluated first and then the next, except you force any operator to evaluate by putting parentheses around it.
The following table explains the precedence of the arithmetic operators:

OPERATORS	**OPERATIONS**	**PRECEDENCE (ORDER OF EVALUATION)**
( )	Parentheses	Evaluated first
*, /, or %	Multiplication, Division, Modulus	Evaluated second. If there are several, they are evaluated from left to right
+ or -	Addition, Subtraction	Evaluated last. If there are several, they are evaluated from left to right

Lets look some examples.
What is the result of 10 + 10 * 5 ?
The answer is 60 not 100. As * has higher precedence than + so 10 * 5 is evaluated first and then the answer 50 is added to 10 and we get the result 60. The answer will be 100 if we force the addition operation to be done first by putting 10 + 10 in parentheses. Thus the same expression rewritten as (10 + 10) * 5 will give the result 100. Note that how the parentheses affect the evaluation of an expression.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Similarly the expression 5 * 3 + 6 / 3 gives the answer 17, and not 7. The evaluation of this expression can be clarified by writing it with the use of parentheses as (5 * 3) + (6 / 3) which gives 15 + 2 = 17. Thus you should be careful while writing arithmetic expressions.

Tips

Use spaces in the coding to make it easy to read and understand
Reserved words can not be used as variable names
There is always a main( ) in a C program that is the starting point of execution
Write one statement per line
Type parentheses '( )' and braces '{ }' in pairs
Use parentheses for clarification in arithmetic expressions
Don't forget semicolon at the end of each statement
C Language is case sensitive so variable names x and X are two different variables

vubookshoppk.com

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Lecture No. 4

Reading Material

Deitel & Deitel – C++ How to Program chapter 1
1.22

Summary

- Sample Program
- Examples of Expressions
- Use of Operators
- Tips

vubookshoppk.com

Sample Program

Problem statement:

Calculate the average age of a class of ten students. Prompt the user to enter the age of each student.

Solution:

Lets first sort out the problem. In the problem we will take the ages of ten students from the user. To store these ages we will use ten variables, one variable for each student's age. We will take the ages of students in whole numbers (in years only, like 10, 12, 15 etc), so we will use the variables of data type **int**. The variables declaration statement in our program will be as follow:

int age1, age2, age3, age4, age5, age6, age7, age8, age9, age10;

We have declared all the ten variables in a single line by using comma separator (,). This is a short method to declare a number of variables of the same data type.
After this we will add all the ages to get the total age and store this total age in a variable. Then we will get the average age of the ten students by dividing this total age by 10. For the storage of total and average ages we need variables. For this purpose we use variable *TotalAge* for the total of ages and *AverageAge* for average of ages respectively.

int TotalAge, AverageAge;

We have declared *AverageAge* as int data type so it can store only whole numbers. The average age of the class can be in real numbers with decimal point (for example if total age is 173 then average age will be 17.3). But the division of integers will

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

produce integer result only and the decimal portion is truncated. If we need the actual result then we should use real numbers (float or double) in our program.

Now we have declared variables for storing different values. In the next step we **prompt** the user to enter the age of first student. We simply show a text line on the screen by using the statement:

cout << "Please enter the age of first student : " ;

So on the screen the sentence "Please enter the age of first student:" will appear. Whenever we are requesting user to enter some information we need to be very clear i.e. write such sentences that are self explanatory and user understands them thoroughly and correctly. Now with the above sentence everyone can understand that age would be entered for the first student. As we are expecting only whole numbers i.e. age in years only i.e. 10, 12 etc, our program is not to expect ages as 13.5 or 12.3 or 12 years and 3 months etc. We can refine our sentence such, that the user understands precisely that the age would be entered in whole number only.

After this we allow the user to enter the age. To, get the *age* entered by the user into a *variable*, we use the statement:

cin >> age1;

Lets have a look on the statement *cin >> age1;* **cin** is the counter part of the **cout**. Here **cin** is the input stream that gets data from the user and assigns it to the variable on its right side. We know that the sign >> indicates the direction of the flow of data. In our statement it means that data comes from user and is assigned to the variable *age1*, where *age1* is a variable used for storing the age entered for *student1*. Similarly we get the ages of all the ten students and store them into respective variables. That means the age of first student in *age1*, the age of second student in *age2* and so on up to 10 students. When **cin** statement is reached in a program, the program stops execution and expects some input from the user. So when *cin >> age1;* is executed, the program expects from the user to type the age of the *student1*. After entering the age, the user has to press the 'enter key'. Pressing 'enter key' conveys to the program that user has finished entering the input and **cin** assigns the input value to the variable on the right hand side which is *age1* in this case. As we have seen earlier that in an assignment statement, we can have only one variable on left hand side of the assignment operator and on right hand side we can have an expression that evaluates to a single value. If we have an expression on the left hand side of assignment operator we get an error i.e. *x = 2 + 4;* is a correct statement but *x + y = 3+ 5;* is an incorrect statement as we can not have an expression on the left hand side. Similarly we can not have an expression after the >> sign with **cin**. So we can have one and only one variable after >> sign i.e. *cin >> x;* is a correct statement and *cin >> x + y;* is an incorrect statement.
Next, we add all these values and store the result to the variable *TotalAge*. We use assignment operator for this purpose. On the right hand side of the assignment operator, we write the expression to add the ages and store the result in the variable, *TotalAge* on left hand side. For this purpose we write the statement as follow:

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

TotalAge = age1 + age2 + age3 + age4 + age5 + age6 + age7 + age8 + age9 + age10 ;

The expression on the right hand side uses many addition operators (+). As these operators have the same precedence, the expression is evaluated from left to right. Thus first *age1* is added to *age2* and then the result of this is added to *age3* and then this result is added to *age4* and so on.
Now we divide this *TotalAge* by 10 and get the average age. We store this average age in the variable i.e. *AverageAge* by writing the statement:

AverageAge = TotalAge / 10;

And at the end we display this average age on the screen by using the following statement:

cout << " The average age of the students is : " << AverageAge;

Here the string enclosed in the quotation marks, will be printed on the screen as it is and the value of *AverageAge* will be printed on the screen.

The complete coding of the program is given below:

vubookshoppk.com

/* This program calculates the average age of a class of ten students after prompting the user to enter the age of each student. */

```
#include <iostream.h>
main ()
{
        // declaration of variables, the age will be in whole numbers
        int age1, age2, age3, age4, age5, age6, age7, age8, age9, age10;
        int TotalAge, AverageAge;

        // take ages of the students from the user
        cout << "Please enter the age of student 1: ";
        cin >> age1;
        cout << "Please enter the age of student 2: ";
        cin >> age2;
        cout << "Please enter the age of student 3: ";
        cin >> age3;
        cout << "Please enter the age of student 4: ";
        cin >> age4;
        cout << "Please enter the age of student 5: ";
        cin >> age5;
        cout << "Please enter the age of student 6: ";
        cin >> age6;
        cout << "Please enter the age of student 7: ";
        cin >> age7;
        cout << "Please enter the age of student 8: ";
        cin >> age8;
```

```
        cout << "Please enter the age of student 9: ";
        cin >> age9;
        cout << "Please enter the age of student 10: ";
        cin >> age10;

        // calculate the total age and average age
        TotalAge = age1 + age2 + age3 + age4 + age5 + age6 + age7 + age8 + age9 +
        age10;
        AverageAge = TotalAge / 10;

        // Display the result ( average age )
        cout << "Average age of class is: " << AverageAge;
}
```

A sample output of the above program is given below.

```
Please enter the age of student 1:  12
Please enter the age of student 2:  13
Please enter the age of student 3:  11
Please enter the age of student 4:  14
Please enter the age of student 5:  13
Please enter the age of student 6:  15
Please enter the age of student 7:  12
Please enter the age of student 8:  13
Please enter the age of student 9:  14
Please enter the age of student 10:  11
Average age of class is: 12
```

In the above output the total age of the students is 123 and the actual average should be 12.3 but as we are using integer data types so the decimal part is truncated and the whole number 12 is assigned to the variable AverageAge.

Examples of Expressions

We have already seen the precedence of arithmetic operators. We have expressions for different calculations in algebraic form, and in our programs we write them in the form of C statements. Let's discuss some more examples to get a better understanding.

We know about the quadratic equation in algebra, that is $y = ax^2 + bx + c$. The quadratic equation in C will be written as $y = a * x * x + b * x + c$. In C, it is not an equation but an assignment statement. We can use parentheses in this statement, this

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

will make the expression statement easy to read and understand. Thus we can rewrite it *as y = a * (x * x) + (b * y) + c.*
Note that we have no power operator in C, just use * to multiply the same value.

Here is another expression in algebra: $x = ax + by + cz^2$. In C the above expression will be as:

$x = a * x + b * y + c * z * z$

The * operator will be evaluated before the + operator. We can rewrite the above statement with the use of parentheses. The same expressions can be written as:

$x = (a * x) + (b * y) + c * (z * z)$

Lets have an other expression in algebra as $x = a(x + b(y + cz^2))$. The parentheses in this equation force the order of evaluation. This expression will be written in C as:

$x = a * (x + b * (y + c * z * z))$

While writing expressions in C we should keep in mind the precedence of the operators and the order of evaluation of the expressions (expressions are evaluated from left to right). Parentheses are used in complicated expressions. In algebra, there may be curly brackets { } and square brackets [ ] in an expression but in C we have only parentheses ( ). Using parentheses, we can make a complex expression easy to read and understand and can force the order of evaluation. We have to be very careful while using parentheses, as parentheses at wrong place can cause an incorrect result. For example, a statement $x = 2 + 4 * 3$ results $x = 14$. As * operator is of higher precedence, $4 * 3$ is evaluated first and then result 12 is added to 4 which gives the result 14. We can rewrite this statement, with the use of parentheses to show it clearly, that multiplication is performed first. Thus we can write it as $x = 2 + (4 * 3)$. But the same statement with different parentheses like $x = (2 + 4) * 3$ will give the result 18, so we have to be careful while using parenthesis and the evaluation order of the expression.

Similarly the equation $(b^2 – 4ac)/2a$ can be written as $(b * b – 4 * a * c) / (2 * a)$. The same statement without using parentheses will be as $b * b – 4 * a * c / 2 * a$. This is wrong as it evaluates to $b^2 – 4ac/2a$ (i.e. 4ac is divided by 2a instead of (b^2-4ac)).

Use of Operators

Here are sample programs which will further explain the use of operators in programming.

Problem Statement:

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Write a program that takes a four digits integer from user and shows the digits on the screen separately i.e. if user enters 7531, it displays 1,3,5,7 separately.

Solution:

Let's first analyze the problem and find out the way how to program it.

Analysis:

First of all, we will sort the problem and find out how we can find digits of an integer. We know that when we divide a number by 10, we get the last digit of the number as remainder. For example when we divide 2415 by 10 we get 5 as remainder. Similarly 3476 divided by 10 gives the remainder 6. We will use this logic in our problem to get the digits of the number. First of all, we declare two variables for storing number and the digit. Let's say that we have a number 1234 to show its digits separately. In our program we will use modulus operator (%) to get the remainder. So we get the first digit of the number 1234 by taking its modulus with 10 (i.e. 1234 % 10). This will give us the digit 4. We will show this digit on the screen by using ***cout*** statement. After this we have to find the next digit. For this we will divide the number by 10 to remove its last digit. Here for example the answer of 1234 divided by 10 is 123.4, we need only three digits and not the decimal part. In C we know that the integer division truncates the decimal part to give the result in whole number only. We will use integer division in our program and declare our variable for storing the number as *int* data type. We will divide the number 1234 by 10 (i.e. 1234 / 10). Thus we will get the number with remaining three digits i.e. 123. Here is a point to be noted that how can we deal with this new number (123)?

There are two ways, **one** is that we declare a new variable of type *int* and assign the value of this new number to it. In this way we have to declare more variables that mean more memory will be used. The **second** way is to reuse the same variable (where number was already stored). As we have seen earlier that we can reassign values to variables like in the statement *x = x + 1*, which means, add 1 to the value of x and assign this resultant value again to x. In this way we are reusing the variable x. We will do the same but use the division operator instead of addition operator according to our need. For this purpose we will write *number = number / 10*. After this statement we have value 123 in the variable *number*.

Again we will get the remainder of this number with the use of modulus operator, dividing the number by 10 (i.e. 123 % 10). Now we will get 3 and display it on the screen. To get the new number with two digits, divide the number by 10. Once again, we get the next digit of the number (i.e. 12) by using the modulus operator with 10, get the digit 2 and display it on the screen. Again get the new number by dividing it by 10 (i.e. 1). We can show it directly, as it is the last digit, or take remainder by using modulus operator with 10. In this way, we get all the digits of the number.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Now let's write the program in C by following the analysis we have made. The complete C program for the above problem is given below. It is easy to understand as we are already familiar with the statements used in it.

```
/* A program that takes a four digits integer from user and shows the digits on the
screen separately i.e. if user enters 7531, it displays 7,5,3,1 separately. */
 #include <iostream.h>
 main()
 {
          // declare variables
          int number, digit;

          // prompt the user for input
          cout << "Please enter 4-digit number:";
          cin >> number;

          // get the first digit and display it on screen
          digit = number % 10;
          cout << "The digits are: ";
          cout << digit << ", ";

          // get the remaining three digits number
          number = number / 10;

          // get the next digit and display it
          digit = number % 10;
          cout << digit << ", ";

          // get the remaining two digits number
          number = number / 10;

          // get the next digit and display it
          digit = number % 10;
          cout << digit << ", ";

          // get the remaining one digit number
          number = number / 10;

          // get the next digit and display it
          digit = number % 10;
          cout << digit;
 }
```

A sample output of the above program is given below.

```
Please enter 4-digit number:  5678
The digits are: 8, 7, 6, 5
```

vubookshoppk.com

Problem Statement:

Write a program that takes radius of a circle from the user and calculates the diameter, circumference and area of the circle and display the result.

Solution:

In this problem we take the input (radius of a circle) from the user. For that we can use *cin* statement to prompt the user to enter the radius of a circle. We store this radius in a variable. We also need other variables to store diameter, circumference and area of the circle. To obtain the correct result, we declare these variables of type **float**, instead of **int** data type, as we know that the **int** data type stores the whole numbers only. Here in our problem the area or circumference of the circle can be in decimal values. After getting the radius we use the formulae to find the diameter, circumference and area of the circle and then display these results on the screen. The solution of this program in coding form is given below.

```
/* Following program takes the radius of a circle from the user and calculates the
diameter, circumference and area of the circle and displays the result. */
#include <iostream.h>
main ()
{
   // declare variables
   float radius, diameter, circumference, area;

   // prompt the user for radius of a circle
   cout << "Please enter the radius of the circle  " ;
   cin >> radius ;

   // calculate the diameter, circumference and area of the circle
   // implementing formula i.e. diameter = 2 r circumference = 2 π r and area = π r2
   diameter = radius * 2 ;
   circumference = 2 * 3.14 * radius ; // 3.14 is the value of π (Pi)
   area = 3.14 * radius * radius ;

   // display the results
   cout << "The diameter of the circle is :   " << diameter ;
   cout << "The circumference of the circle is :   " << circumference ;
   cout << "The area of the circle is :   " << area ;
 }
```

A sample output of the above program is given below.

```
Please enter the radius of the circle    5
The diameter of the circle is :  10
The circumference of the circle is :   31.4
The area of the circle is :   78.5
```

vubookshoppk.com

Tips

- Use descriptive names for variables
- Indent the code for better readability and understanding
- Use parenthesis for clarity and to force the order of evaluation in an expression
- Reuse the variables for better usage of memory
- Take care of division by zero
- Analyze the problem properly, and then start coding (i.e. first think and then write)

vubookshoppk.com

Lecture No. 5

Reading Material

Deitel & Deitel – C++ How to Program	chapter 2 2.4, 2.5, 2.6, 2.19, 2.20

Summary

- Conditional Statements
- Flow Charting
  - Sample Program 1
- if/else structure
- Logical Operators
  - Sample Program 2
- Tips

vubookshoppk.com

Conditional Statements (Decision Making)

In every day life, we are often making decisions. We perform different tasks while taking decisions. For example, the statement 'if the milk shop is open, bring one liter of milk while returning home from college', involves this phenomenon.
In this statement, there is an element of decision making. We bring one litre of milk if the shop is open. And if the shop is closed, we come back to home without milk.
Thus we are making a decision on the condition that the shop is open. The decision-making process is everywhere in our daily life. We see that the college gives admission to a student if he has the required percentage in his previous examination and/or in the entry test. Similarly administration of a basketball team of the college decides that the students having height more than six feet can be members of the team.
In the previous lectures, we have written simple elementary programs. For writing interesting and useful programs, we have to introduce the decision making power in them. Now we will see what kind of decisions are there in programming and how these can be used.
Every programming language provides a structure for decision making.
'C' also provides this structure. The statement used for decisions in 'C' language is known as the '*if* statement'. The *if* statement has a simple structure. That is

```
if ( condition )
    Statement (or group of statements)
```

The above statements mean, If condition is true, then execute the statement or a group of statements. Here the condition is a statement which explains the condition on which a decision will be made. We can understand it from the example that Ali can

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

become the member of the basket ball team if he has a height more than six feet .In this case, the condition will be

if (Ali's height is greater than six feet)
Ali can be a member of team

We have written the condition in English language. Now let's see how we can implement this in terms of variables, operators and C statements. In the program, we will write the condition in parentheses, followed by a statement or group of statements to be executed.
Now here is the concept of block of statements. We use braces { } to make a group (block) of a number of statements. We put '{' before first statement and '}' after the last statement. Thus if we have to do many things after the *if* statement. The structure of *if* statement becomes as under

```
if (condition)
{
   statement;
   statement;
   .
   .
  statement;
}
```

Note the indentation of the lines and semi-colon after each statement. Semi-colons are necessary after every C statement. The indentation is only a matter of style. It makes the code easy to read and understand from where a block starts, ends and what kind of block it is. It does not affect the logic of the program. But the braces can affect the logic. We can also write a comment line to state the purpose of code block.
Let's consider a simple example to explain the *if* statement. Suppose, we have ages of two students (say for the time being we have got these ages in variables). These variables are- age1 and age2. Now we say that if the age1 is greater than age2, then display the statement 'Student 1 is older than student 2'.
The coding for this program will be as below

```
#include <iostream.h>
main()
{
        int age1, age2;
        age1 = 12;
        age2 = 10;
        if (age1 > age2)
            cout << "Student 1 is older than student 2";
}
```

Here, in our code we see a new operator i.e. ' > ' (greater than) in the *if* statement. We need such operators (like greater than, less than, equal to etc) while making decisions. These operators are called 'relational operators'. These are almost the same relational operators we use in algebra. Following table summarizes the relational operators.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

	Algebraic	In C language	Example	Meaning
Greater than	>	>	x > y	x is greater than y
Equal to	=	==	x == y	x is equal to y
Less than	<	<	x < y	x is less than y
Greater than or equal to	≥	>=	x >= y	x is greater than or equal to y
Less than or equal to	≤	<=	x <= y	x is less than or equal to y
Not equal to	≠	!=	x != y	x is not equal to y

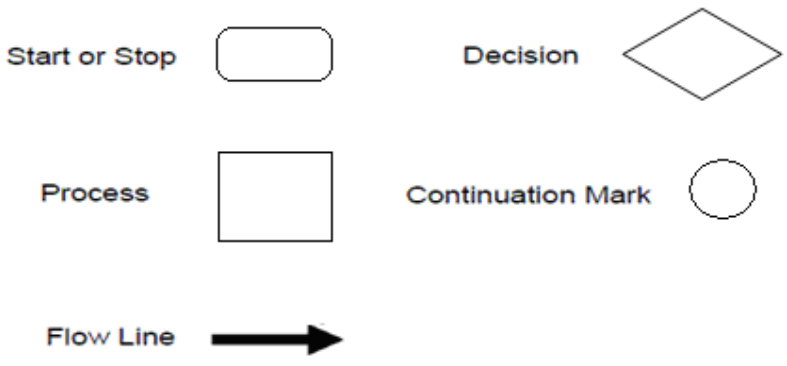
Note that there is no space between ==, >=, <= and !=.
These are considered as single operators.
The operator == (equal to) is different from the operator =. We know that operator = is the assignment operator which is used in assignment statement to assign a value to a variable.
Don't confuse the assignment operator (=) with equal to operator (==). If we write single = in condition of *if* statement. For example, if we write *if* (x = 2), the compiler will not give error. This means that it is not a syntax error. The conditional expression in *if* statement returns a value. In this case, x = 2 will also have some value but it will not in the form of true or false. So it will create a logical error. So be careful while using equal to condition in *if* statement.

Flow Charting

There are different techniques that are used to analyze and design a program. We will use the flow chart technique. A flow chart is a pictorial representation of a program. There are labeled geometrical symbols, together with the arrows connecting one symbol with other.
A flow chart helps in correctly designing the program by visually showing the sequence of instructions to be executed. A programmer can trace and rectify the logical errors by first drawing a flow chart and then simulating it.

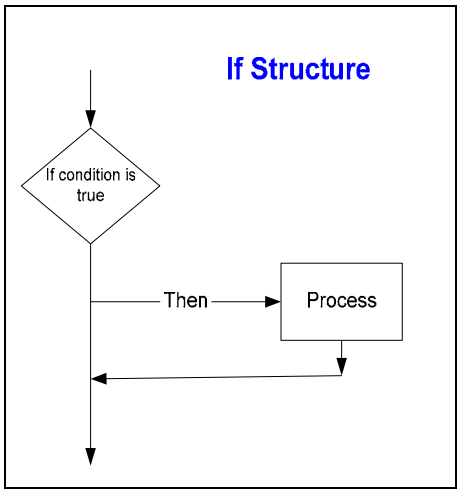
Flow Chart Symbols

Below are some of the main symbols used in the flow chart.



All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

The flow chart for the *if* structure is shown in the figure below.



Sample Program 1

Now let's see the usage of relational operators by an example. There are two students Amer and Amara. We take their ages from the user, compare them and tell who is older?

As there are two students to be compared in terms of age, we need to declare two variables to store their ages. We declare two variables AmerAge and AmaraAge of type int. The variable names are one continuous word as we can't use spaces in a variable name.

Here is an important point about variables declaration. We should assign an initial value (preferably 0 for integers) to variables when we declare them. This is called initialization of variables.

We can do this in one line while declaring a variable like *int x = 0*; This statement will declare a variable of name x with data type int and will assign a value 0 to this variable. Initializing a variable in this way is just a matter of style. You can initialize a variable on a separate line after declaring it. It is a good programming practice to initialize a variable.

Now we prompt the user to enter Amer's age and store it into variable AmerAge. Then similarly we get Amara's age from the user in the variable AmaraAge.

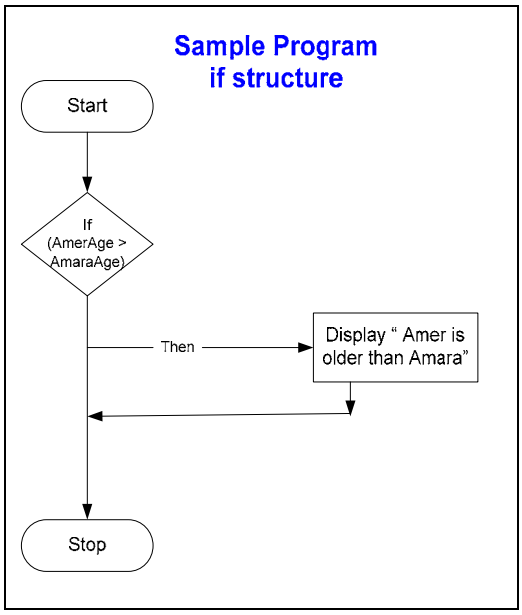
While comparing the ages, we will use the *if* statement to see whether Amer's age is greater than Amara's. We will use > (greater than) operator to compare the ages. This can be written as *if* (AmerAge > AmaraAge) .

With this *if* statement, we write the statement cout << "Amer is greater than Amara" ;

It's a simple one line test i.e. 'if Amer's age is greater than Amara's', then display the message 'Amer is older than Amara'.

The flow chart for the above problem is as under.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533



The complete code of the program is given below.

```
/* This program test that if the age of Amer is greater than Amara's age and displays the result.
*/

# include <iostream.h>

main ( )
{
        int AmerAge, AmaraAge;
        //prompt the user to enter Amer's age
        cout << "Please enter Amer's age     " ;
        cin >> AmerAge;
        //prompt the user to enter Amara's age
        cout << "Please enter Amara's age     " ;
        cin >> AmaraAge;
        //perform the test
        if (AmerAge > AmaraAge )
                cout << " Amer is older than Amara";
}
```

vubookshoppk.com

In our program, we write a single statement with the *if* condition. This statement executes if the condition is true. If we want to execute more than one statements, then we have to enclose all these statements in curly brackets { }. This comprises a block of statements which will execute depending upon the condition. This block may contain a single statement just like in our problem. So we can write the *if* statement as follow.

```
        if (AmerAge > AmaraAge )
        {
                cout << " Amer is older than Amara";
        }
```

A sample execution of the program provides the following output.

```
Please enter Amer's age        16
Please enter Amara's age       14
 Amer is older than Amara
```

Now think what happens if the condition in the *if* statement is not true i.e. Amer's age is not greater than Amara's. In this case, if the user enters Amer's age less than Amara's, then our program does nothing. So to check this condition, another *if* statement after the first *if* statement is required. Then our program will be as:

```
/* This program checks the age of Amer and  Amara's and
displays the appropriate  the message. The program is using
two if statements.*/

# include <iostream.h>

main ( )
{
        int AmerAge, AmaraAge;
        //prompt the user to enter Amer's age
        cout << "Please enter Amer's age     " ;
        cin >> AmerAge;
        //prompt the user to enter Amara's age
        cout << "Please enter Amara's age     " ;
        cin >> AmaraAge;
        //perform the test
        if (AmerAge > AmaraAge )
           {
                cout << " Amer is older than Amara";
           }
        if (AmerAge < AmaraAge )
            {
                cout << " Amer is younger than Amara";
            }
}
```

Now our program decides properly about the ages entered by the user.
After getting ages from the user, the *if* statements are tested and if statement will be executed if the condition evaluates to true.

If/else Structure

We have seen that the *if* structure executes its block of statement(s) only when the condition is true, otherwise the statements are skipped. The if/else structure allows the programmer to specify that a different block of statement(s) is to be executed when the condition is false. The structure of if/else selection is as follows.

if (condition)

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
{
        statement(s);
}
else
{
        statement(s);
}
```

Thus using this structure we can write the construct of our program as

```
if (AmerAge > AmaraAge )
{
        cout << " Amer is older than Amara";
}
else
{
        cout << " Amer is younger than Amara";
}
```

In this construct, the program checks the condition in *if* statement .If the condition is true, then the line "Amer is greater than Amara" is printed. Otherwise (if condition is not true), the statement related to *else* is executed and the message "Amer is younger than Amara" is printed. Here in if/else structure an important thing is that the else part is executed for all the cases (conditions) other than the case which is stated in the *if* condition.

And in the comparison, we know that there are three conditions i.e. first value is greater than the second value, first value is less than the second value and first value is equal to the second value. Here in the above program construct the else part competes the greater than conditions and covers both less than and equal to conditions.

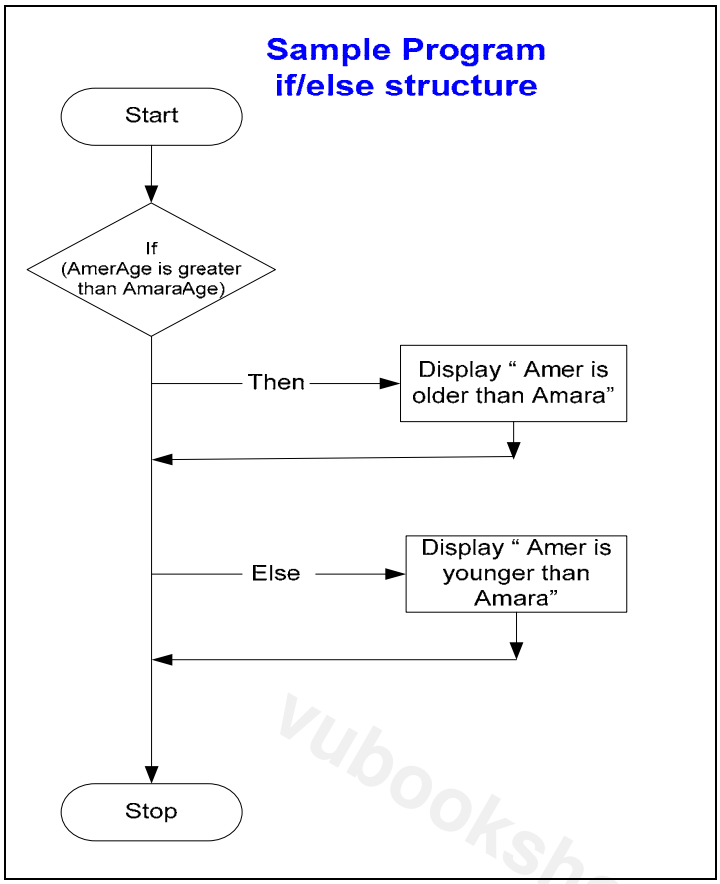
Thus in the above program construct, the message "Amer is younger than Amara" is displayed even if Amer's age is the same as Amara's age. This is logically incorrect and so to make this correct, we should display the message "Amer is younger than or is of the same age as Amara". Now this statement describes both the cases other than the one 'Amer is greater than Amara'.

The use of else saves us from writing different *if* statements to compare different conditions, in this way it cover the range of checks to complete the comparison.

If we want to state the condition "Amer is greater than or is of the same age as Amara's" then we use the greater than or equal to operator (i.e. >=) in the *if* statement and less than operator (<) in the else statement to complete the comparison.

It is very important to check all the conditions while making decisions for good, complete and logical results. Make sure that all cases are covered and there is no such case in which the program does not respond.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533



vubookshoppk.com

Logical Operators

There are many occasions when we face complex conditions to make a decision. This means that a decision depends upon more than one condition in different ways. Here we combine the conditions with AND or OR. For example, a boy can be selected in basket ball team only if he is more than 18 years old and has a height of 6 feet. In this statement a boy who wants to be selected in the basket ball team must have both the conditions fulfilled. This means that AND forces both the conditions to be true. Similarly we say that a person can be admitted to the university if he has a BCS degree OR BSC degree. In this statement, it is clear that a person will be admitted to the university if he has any one of the two degrees.

In programming we use logical operators (&& and ||) for AND and OR respectively with relational operators. These are binary operators and take two operands. These operators use logical expressions as operands, which return TRUE or FALSE.

The following table (called truth table) can be used to get the result of the && operator and || operator with possible values of their operands. It is used to explain the result obtained by the && and || operators.

Expression 1	**Expression 2**	**Expression 1 && Expression 2**	**Expression 1 || Expression 2**
True	False	false	True

True	True	true	True
False	False	false	False
False	True	false	True

The && operator has a higher precedence than the || operator. Both operators associate from left to right. An expressions containing && or || is evaluated only until truth or falsehood is known. Thus evaluation of the expression *(age > 18) && (height > 6)* will stop immediately if *age > 18* is false (i.e. the entire expression is false) and continue if *age > 18* is true (i.e. the entire expression could still be true if the condition height > 6 is true).

There is another logical operator that is called logical negation. The sign ! is used for this operator. This operand enables a programmer to 'reverse' the meaning of a condition. This is a unary operator that has only a single condition as an operand. The operator ! is placed before a condition. If the original condition (without the ! operator) is **false** then the ! operator before it converts it to **true** and the statements attached to this are executed.
Look at the following expression

```
if ( ! (age > 18 ))
    cout << " The age is less than 18";
```

Here the **cout** statement will be executed if the original condition (age > 18) is false because the ! operator before it reverses this **false** to **true**.

The truth table for the logical negation operator (!) is given below.

Expression	**! Expression**
True	False
False	True

vubookshoppk.com

Sample Program 2

Problem statement

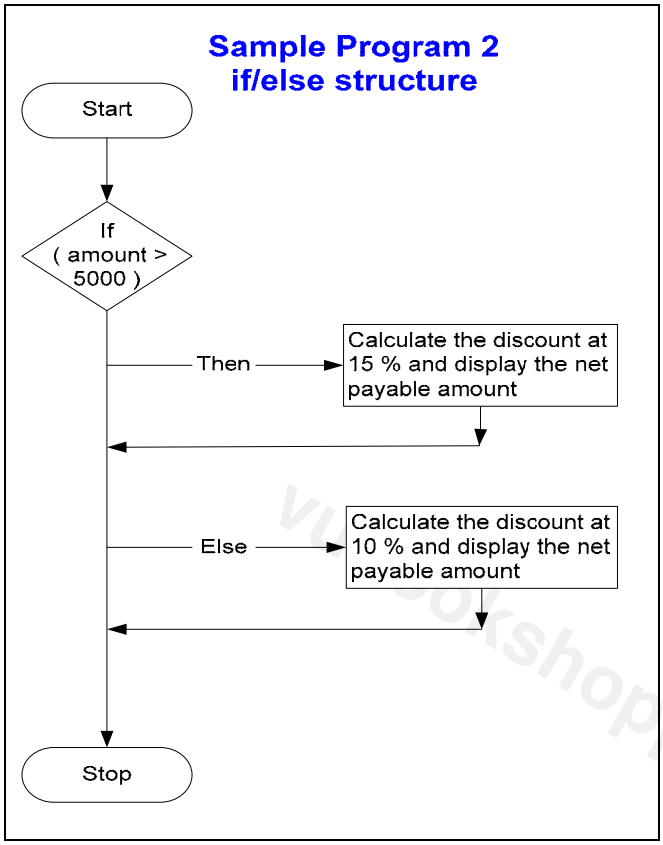
A shopkeeper announces a package for customers that he will give 10 % discount on all bills and if a bill amount is greater than 5000 then a discount of 15 %. Write a C program which takes amount of the bill from user and calculates the payable amount by applying the above discount criteria and display it on the screen.

Solution

In this problem we are going to make decision on the basis of the bill amount, so we will be using *if* statement. We declare three variables amount, discount and netPayable and initialize them. Next we prompt the user to enter the amount of the bill. After this we implement the *if statement* to test the amount entered by the user. As we see in the problem statement that if the amount is greater than 5000 then the discount rate is 15 % otherwise (i.e. the amount is less than or equal to 5000) the

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

discount rate is 10 %. So we check the amount in *if statement*. If it is greater than 5000 then the condition is true then the if block is executed otherwise if amount is not greater than 5000 then the else block is executed.
The analysis and the flow of the program is shown by the following flow chart.



vubookshoppk.com

The complete program code is given below:

```
/* This program calculates the discount amount for a customer. As different discount
percentage applies on different amount so program is using if statement for deciding
which discount is applicable and display the result. */

# include <iostream.h>

main ( )
{
        double amount, discount, netPayable ;
        amount = 0 ;
        netPayable = 0 ;
        discount = 0 ;
        // prompt the user to enter the bill amount
        cout << "Please enter the amount of the bill     " ;
        cin >> amount ;
        //test the conditions and calculate net payable

        if ( amount > 5000 )
          {
             //calculate amount at 15 % discount
            discount = amount * (15.0 / 100);
            netPayable = amount - discount;
            cout << "The discount at the rate 15 % is Rupees     " << discount << endl;
            cout << "The payable amount is Rupees    " << netPayable ;
          }
       else
          {
             // calculate amount at 10 % discount
           discount = amount * (10.0 / 100);
           netPayable = amount - discount;
           cout << "The discount at the rate 10 % is Rupees     " << discount << endl ;
            cout << "The payable amount is Rupees    " << netPayable ;
          }

}
```

In the program we declared the variables as *double*. We do this to get the correct results (results may be in decimal points) of the calculations. Look at the statement which calculates the discount. The statement is

discount = amount * (15.0 / 100) ;

Here in the above statement we write **15.0** instead of **15**. If we write here **15** then the division **15 / 100** will be evaluated as integer division and the result of division (0.15) will be truncated and we get 0 and this will result the whole calculation to zero. So it is necessary to write at least one operand in decimal form to get the correct result by division and we should also declare the variables as **float** or **double**. We do the same in the line discount = amount * (10.0 / 100);

A sample execution of the program is given below

vubookshoppk.com

```
Please enter the amount of the bill        6500
The discount at the rate 15 % is Rupees        975
The payable amount is Rupees      5525
```

Tips

- Always put the braces in an **if/else** structure
- Type the beginning and ending braces before typing inside them
- Indent both body statements of an *if* and *else* structure
- Be careful while combining the conditions with logical operators
- Use **if/else** structure instead of a number of single selection if statements

vubookshoppk.com

Lecture No. 6

Reading Material

Deitel & Deitel – C++ How to Program chapter 2
2.7, 2.8, 2.9, 2.20

Summary
Repetition Structure (Loop)
Overflow Condition
Sample Program 1
Sample Program 2
Infinite Loop
Properties of While loop
Flow Chart
Sample Program 3
Tips

Repetition Structure (Loop)

In our day to day life, most of the things are repeated. Days and nights repeat themselves 30 times a month. Four seasons replace each other every year. We can see similar phenomenon in the practical life. For example, in the payroll system, some procedures are same for all the employees. These are repeatedly applied while dealing with the employees. So repetition is very useful structure in the programming.

Let's discuss a problem to understand it thoroughly. We have to calculate the sum of first 10 whole numbers i.e. add the numbers from 1 to 10. Following statement may be one way to do it.

```
cout << "Sum of first 10 numbers is = " << 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9  +
10;
```

This method is perfectly fine as the syntax is right. The answer is also correct. This procedure can also be adopted while calculating the sum of numbers from 1 to 100. We can write the above statement adding all the digits from 1 to 100. But this method will not be suitable for computing the sum of numbers from 1 to 1000.The addition of a very big number of digits will result in a very ugly and boring statement. Let's analyze it carefully. Our first integer is 1, is there any other way to find out what is the next integer? Yes, we can add 1 to the integer and get the next integer which is 2. To find the next integer (i.e. 3) we add 1 to the previous integer (i.e. 2) and get the next integer which is 3. So whenever we have to find out the next integer, we have to add 1 to the previous integer.

We have to calculate the sum of first 1000 integers by taking a variable *sum* of type *int*. It is a good programming practice to initialize the variable before using it. Here, we initialize the variable *sum* with zero.

```
int sum = 0;
```

Now we get the first integer i.e. 1. We add this to the *sum* (*sum* becomes 0 + 1 = 1). Now get the next integer which can be obtained by adding 1 to the previous integer i.e. 2 and add it to the *sum* (*sum* becomes 1 + 2 = 3). Get the next integer by adding 1 to the previous integer and add it to the *sum* (*sum* becomes 3 + 3 = 6) and so on.

This way, we get the next integer by adding 1 to the previous integer and the

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

new integer to the sum. It is obvious that we are repeating this procedure again and again i.e. adding 1 to the previous integer and add this new integer to the sum. So we need some repetition structure in the programming language. There are many looping constructs in C Language. The repetition structure we are discussing in this lecture is 'while loop structure'. '*while'* is also a key word of 'C' so it cannot be used as a variable name.

While means, 'do it until the condition is true'. The use of while construct can be helpful in repeating a set of instructions under some condition. We can also use curly braces with *while* just like we used with *if*. If we omit to use the braces with *while* construct, then only one statement after while will be repeatedly executed. For good programming practices, always use braces with *while* irrespective of the number of statements in *while* block. The code will also be indented inside the *while* block as Indentation makes the code easy to understand.

The syntax of *while* construct is as under:

```
while ( Logical Expression ) {
        statement1;
         statement2;
         ……….
 }
```

The logical expression contains a logical or relational operator. While this logical expression is true, the statements will be executed repeatedly. When this logical expression becomes false, the statements within the *while* block, will not be executed. Rather the next statement in the program after *while* block, will be executed.

Let's discuss again the same problem i.e. calculation of the sum of first 1000 integers starting from 1. For this purpose, we need a variable to store the sum of integers and declare a variable named *sum*. Always use the self explanatory variable names. The declaration of the variable sum in this case is:

```
int sum = 0;
```

The above statement has performed two tasks i.e. it declared the variable *sum* of type *int* and also initialized it with zero. As it is good programming practice to initialize all the variables when declared, the above statement can be written as:

```
int sum;
sum = 0;
```

Here we need a variable to store numbers. So we declare a variable *number* of type *int*. This variable will be used to store integers.

```
int number;
```

As we have declared another variable of *int* data type, so the variables of same data type can be declared in one line.

```
int sum, number;
```

Going back to our problem, we need to sum up all the integers from 1 to 1000. Our first integer is 1. The variable *number* is to be used to store integers, so we will initialize it by 1 as our first integer is 1:

```
number = 1;
```

Now we have two variables- *sum* and *number. That means* we have two memory locations labeled as sum and number which will be used to store sum of integers and integers respectively. In the variable *sum,* we have to add all the integers from 1 to

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

1000. So we will add the value of variable *number* into variable *sum,* till the time the value of number becomes 1000. So when the value of number becomes 1000, we will stop adding integers into *sum.* It will become the condition of our while loop. We can say sum the integers until integer becomes 1000. In C language, this condition can be written as:

```
while ( number  <= 1000 ) {
  ……Action ……
}
```

The above condition means, 'perform the action until the number is 1000 or less than 1000'. What will be the Action? Add the *number*, the value of number is 1 initially, into sum. This is a very simple statement:

```
sum = sum  + number;
```

Let's analyze the above statement carefully. We did not write *sum = number;* as this statement will replace the contents of *sum* and the previous value of *sum* will be wasted as this is an assignment statement. What we did? We added the contents of *sum* and contents of *number* first (i.e. 0 + 1) and then stored the result of this (i.e. 1) to the *sum*.

Now we need to generate next integer and add it to the sum. How can we get the next integer? Just by adding 1 to the integer, we will get the next integer. In 'C', we will write it as:

```
number = number + 1;
```

Similarly in the above statement, we get the original contents of number (i.e. 1). Add 1 to them and then store the result (i.e. 2) into the number. Now we need to add this new number into sum:

```
sum = sum + number;
```

We add the contents of *sum* (i.e. 1) to the contents of number (i.e. 1) and then store the result (i.e. 2) to the sum. Again we need to get the next integer which can be obtained by adding 1 to the number. In other words, our action consists of only two statements i.e. add the number to the sum and get the next integer. So our action statements will be:

```
sum = sum + number;
number = number + 1;
```

Putting the action statements in *while construct:*

```
while ( number  <= 1000 ) {
     sum = sum + number;
    number = number + 1;
}
```

Let's analyze the above while loop. Initially the contents of *number* is 1. The condition in *while* loop (i.e. *number <= 1000*) will be evaluated as true, contents of *sum* and contents of *number* will be added and the result will be stored into *sum*. Now 1 will be added to the contents of *number* and *number* becomes 2. Again the condition in *while* loop will be evaluated as true and the contents of *sum* will be added to the contents of *number* .The result will be stored into *sum*. Next 1 will be added to the contents of *number* and *number* becomes 3 and so on. When *number* becomes 1000, the condition in *while* loop evaluates to be true, as we have used <= (less than or equal

vubookshoppk.com

to) in the condition. The contents of *sum* will be added to the contents of *number* (i.e. 1000) and the result will be stored into the *sum*. Next 1 will be added to the contents of *number* and *number* becomes 1001. Now the condition in *while* loop is evaluated to false, as *number* is no more less than or equal to 1000 (i.e. *number* has become 1001). When the condition of *while* loop becomes false, loop is terminated. The control of the program will go to the next statement following the ending brace of the *while* construct. After the *while* construct, we can display the result using the *cout* statement.

cout << " The sum of first 1000 integers starting from 1 is " << sum;

The complete code of the program is as follows:

```
/* This program calculate the sum of first 1000 integers */
#include <iostream.h>

main()
{
   //declaration of variables
   int sum, number;

   //Initialization of the variables
   sum = 0;
   number = 1;

   // using the while loop to find out the sum of first 1000 integers starting from 1

   while(number <= 1000)
   {
      // Adding the integer to the contents of sum
      sum = sum + number;

      // Generate the next integer by adding 1 to the integer
      number = number + 1;
   }

   cout << "The sum of first 1000 integers starting from 1 is " << sum;
}
```

The output of the program is:

```
The sum of first 1000 integers starting from 1 is 500500
```

While construct is a very elegant and powerful construct. We have seen that it is very easy to sum first 1000 integers just with three statements. Suppose we have to calculate the sum of first 20000 integers. How can we do that? We just have to change the condition in the while loop (i.e. number <= 20000).

Overflow Condition:

We can change this condition to 10000 or even more. Just try some more numbers. How far can you go with the limit? We know that integers are allocated a fixed space in memory (i.e. 32 bits in most PCs) and we can not store a number which requires more bits than integer, into a variable of data type, *int*. If the sum of integers becomes larger than this limit (i.e. sum of integers becomes larger than 32 bits can store), two things can happen here. The program will give an error during execution, compiler can not detect such errors. These errors are known as run time errors. The second thing is that 32 bits of the result will be stored and extra bits will be wasted, so our result will not be correct as we have wasted the information. This is called overflow.

vubookshoppk.com

When we try to store larger information in, than a data type can store, overflow condition occurs. When overflow condition occurs either a run-time error is generated or wrong value is stored.

Sample Program 1:

To calculate the sum of 2000 integers, we will change the program (i.e. the while condition) in the editor and compile it and run it again. If we need to calculate the sum of first 5000 integers, we will change the program again in the editor and compile and run it again. We are doing this work again in a loop. Change the program in the editor, compile, execute it, again change the program, compile and execute it and so on. Are we doing this in a loop? We can make our program more intelligent so that we don't need to change the condition every time. We can modify the condition as:

```
int upperLimit;
 while (number <= upperLimit)
```

where upperLimit is a variable of data type int. When the value of upperLimit is 1000, the program will calculate the sum of first 1000 integers. When the value of upperLimit is 5000, the program will calculate the sum of first 5000 integers. Now we can make it re-usable and more effective by requesting the user to enter the value for upper limit:

```
cout << "Please enter the upper limit for which you want the sum ";
cin >> upperLimit;
```

We don't have to change our program every time when the limit changes. For the sum of integers, this program has become generic. We can calculate the sum of any number of integers without changing the program. To make the display statement more understandable, we can change our *cout* statement as:

```
cout << " The sum of  first " << upperLimit << " integers is " << sum;
```

Sample Program 2:

Problem statement:

Calculate the sum of even numbers for a given upper limit of integers.

Solution:

We analyze the problem and know that while statement will be used. We need to sum even numbers only. How can we decide that a number is even or not? We know that the number that is divisible by 2 is an even number. How can we do this in C language? We can say that if a number is divisible by 2, it means its remainder is zero, when divided by 2. To get a remainder we can use C's modulus operator i.e. %. We can say that for a number if the expression (number % 2) results in zero, the number is even. Putting this in a conditional statement:

```
If ( ( number % 2)  == 0 )
```

The above conditional statement becomes true, when the number is even and false when the number is odd (A number is either even or odd).

The complete code of the program is as follows:

```
/* This program calculates sum of even numbers for a given upper limit of
integers */
#include <iostream.h>

main()
{
   //declaration of variables
   int sum, number, upperLimit;

   //Initialization of the variables
   sum = 0;
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
    number = 1;

    // Prompt the user to enter upper limit of integers
    cout << “Please enter the upper limit for which you want the sum ” ;
    cin  >> upperLimit;

    // using the while loop to find out the sum of first 1000 integers starting from 1

    while(number <= upperLimit)
    {
        // Adding the even integer to the contents of sum
        if ( ( number % 2 ) == 0 )
        {
            sum = sum + number;
        }

       // Generate the next integer by adding 1 to the integer
       number = number + 1;
    }

    cout << "The sum of even numbers of first “ << upperLimit << “ integers starting
from 1 is " << sum;
}
```

The output of the program is:

```
Please enter the upper limit for which you want the sum 10
The sum of even numbers of first 10 integers starting from 1 is 30
```

Suppose if we don’t have modulus operator in the C language. Is there any other way to find out the even numbers? We know that in C integer division gives the integer result and the decimal portion is truncated. So the expression (2 * (number / 2)) gives the number as a result, if the number is even only. So we can change our condition in *if* statement as:

```
if ( ( 2 * ( number /2 ) ) == number )
```

Infinite Loop:

Consider the condition in the *while* structure that is (number <= upperLimit) and in the *while* block the value of *number* is changing (number = number + 1) to ensure that the condition is tested again next time. If it is true, the *while* block is executed and so on. So in the *while* block statements, the variable used in condition must change its value so that we have some definite number of repetitions. What will happen if we do not write the statement *number = number + 1;* in our program? The value of *number* will not change, so the condition in the *while* loop will be true always and the loop will be executed forever. Such loops in which the condition is always true are known as infinite loops as there are infinite repetitions in it.

Property of while loop:

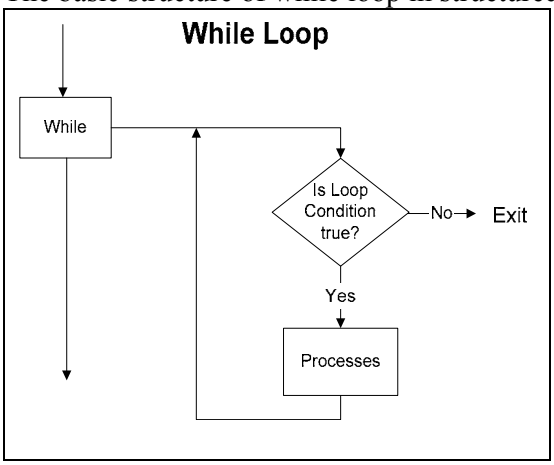
In the above example, if the user enters 0, as the value for upper limit. In the while condition we test (number <= upperLimit) i.e. number is less than or equal to upperLimit (0), this test return false. The control of the program will go to the next statement after the *while* block. The statements in *while* structure will not be executed even for a single time. So the property of while loop is that it may execute zero or more time.

The *while* loop is terminated, when the condition is tested as false. Make sure that the loop test has an adequate exit. Always use braces for the loop structure. If you forget

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

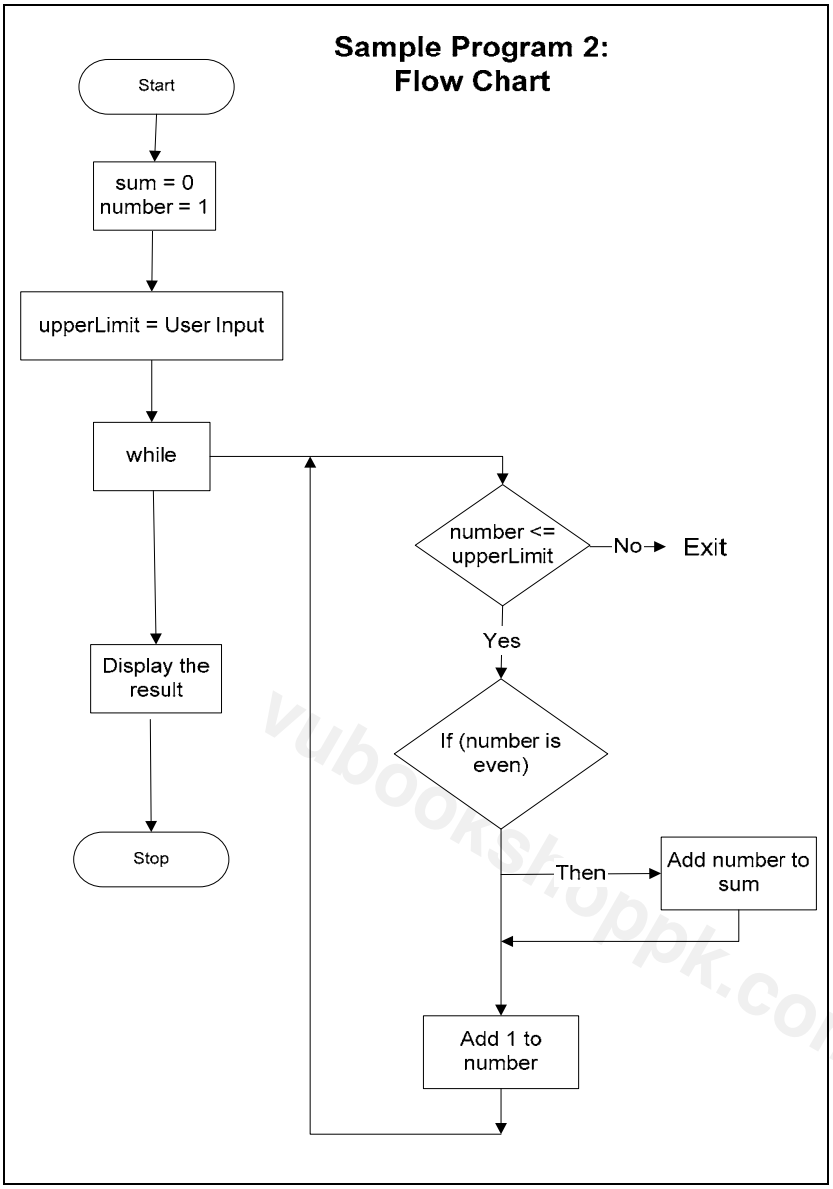
to put the braces, only one statement after the *while* statement is considered in the *while* block.

Flow Chart:
The basic structure of while loop in structured flow chart is:



At first, we will draw a rectangle and write while in it. Then draw a line to its right and use the decision symbol i.e. diamond diagram. Write the loop condition in the diamond and draw a line down to diamond which represents the flow when the decision is true. All the repeated processes are drawn here using rectangles. Then a line is drawn from the last process going back to the while and decision connection line. We have a line on the right side of diamond which is the exit of while loop. The *while* loop terminates, when the loop condition evaluates to false and the control gets out of *while* structure.
Here is the flow chart for sample program 2:

vubookshoppk.com



So far, we have been drawing flow charts after coding the program but actually we have to draw the flow chart first and then start coding.

Sample Program 3:

Problem statement:

Calculate the factorial of a given number.

Solution:

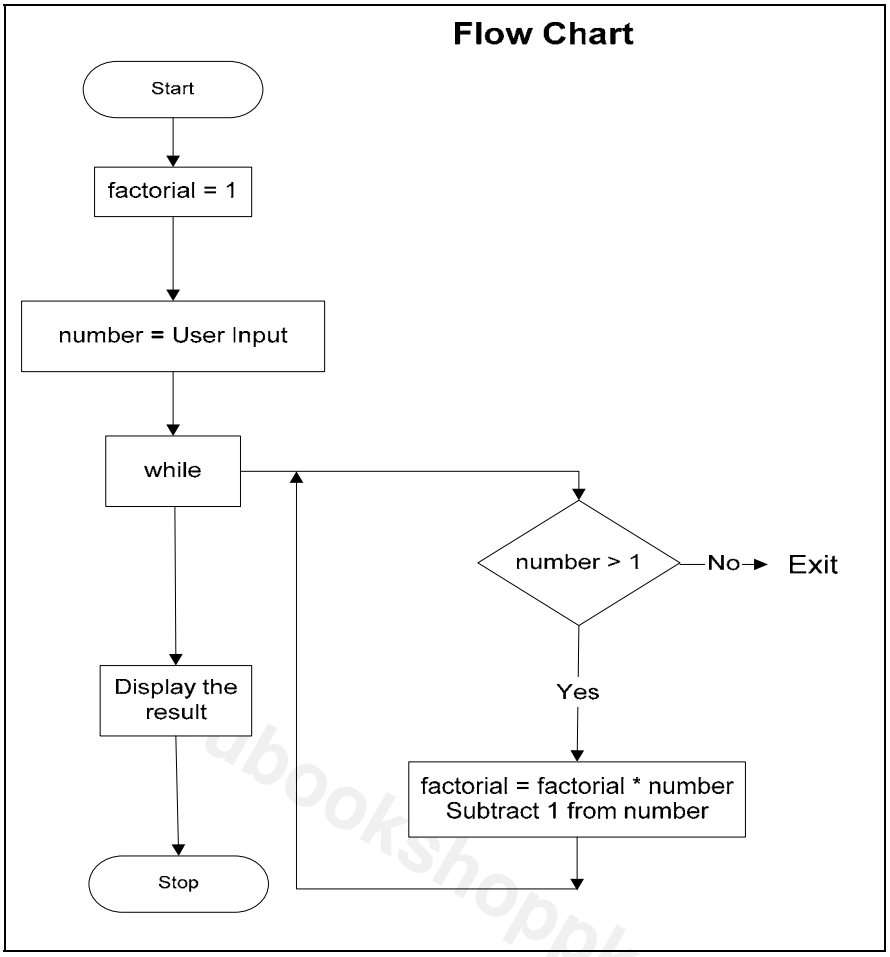
The factorial of a number N is defined as:

N(N-1)(N-2)…….3.2.1

By looking at the problem, we can see that there is a repetition of multiplication of numbers. A loop is needed to write a program to solve a factorial of a number. Let's think in terms of writing a generic program to calculate the factorial so that we can get the factorial of any number. We have to multiply the number with the next decremented number until the number becomes 1. So the value of number will decrease by 1 in each repetition.

Here is the flow chart for the factorial.

vubookshoppk.com



Here is the code of the program.

```
/*This program calculates the factorial of a given number.*/

#include <iostream.h>

main()
{
   //declaration of variables
   int factorial, number;

   //Initialization of the variables
   factorial = 1;
   number = 1;

   // Prompt the user to enter upper limit of integers
   cout << “Please enter the number for factorial ” ;
   cin  >> number;

   // using the while loop to find out the factorial

   while(number > 1)
   {
      factorial = factorial * number;
      number = number - 1;
   }

   cout << "The factorial is  “ << factorial;
```

vubookshoppk.com

```
}
```

Exercise:
Calculate the sum of odd integers for a given upper limit. Also draw flow chart of the program.
Calculate the sum of even and odd integers separately for a given upper limit using only one loop structure. Also draw flow chart of the program.

Tips
Always use the self explanatory variable names
Practice a lot. Practice makes a man perfect
While loop may execute zero or more time
Make sure that loop test (condition) has an acceptable exit.

vubookshoppk.com

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Lecture No. 7

Reading Material

Deitel & Deitel – C++ How to Program Chapter 2

2.11, 2.12, 2.14, 2.15, 2.17

Summary

Do-While Statement
Example
for Statement
Sample Program 1
Increment/decrement Operators
Sample Program 2
Tips

vubookshoppk.com

Do-While Statement

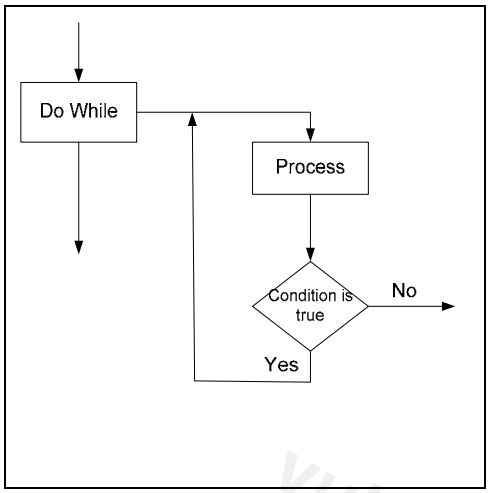
We have seen that there may be certain situations when the body of *while loop* does not execute even a single time. This occurs when the condition in *while* is false. In *while* loop, the condition is tested first and the statements in the body are executed only when this condition is true. If the condition is false, then the control goes directly to the statement after the closed brace of the *while loop*. So we can say that in *while* structure, the loop can execute zero or more times. There may be situations where we may need that some task must be performed at least once.

For example, a computer program has a character stored from a-z. It gives to user five chances or tries to guess the character. In this case, the task of guessing the character must be performed at least once. To ensure that a block of statements is executed at least once, C provides a *do-while* structure. The syntax of *do-while* structure is as under:

```
do
 {
   statement(s);

 }
 while ( condition ) ;
```

Here we see that the condition is tested after executing the statements of the loop body. Thus, the loop body is executed at least once and then the condition in *do while statement* is tested. If it is true, the execution of the loop body is repeated. In case, it proves otherwise (i.e. false), then the control goes to the statement next to the *do*

while statement. This structure describes ‘execute the statements enclosed in braces in *do* clause' when the condition in *while* clause is true.
Broadly speaking, in *while* loop, the condition is tested at the beginning of the loop before the body of the loop is performed. Whereas in *do-while* loop, the condition is tested after the loop body is performed.
Therefore, in *do-while* loop, the body of the loop is executed at least once.
The flow chart of do-while structure is as follow:



Example

Let’s consider the example of guessing a character. We have a character in the program to be guessed by the user. Let’s call it ‘z’. The program allows five tries (chances) to the user to guess the character. We declare a variable **tryNum** to store the number of tries. The program prompts the user to enter a character for guessing. We store this character in a variable **c.**
We declare the variable **c** of type *char*. The data type *char* is used to store a single character. We assign a character to a variable of *char* type by putting the character in single quotes. Thus the assignment statement to assign a value to a *char* variable will be as **c = ‘a’**. Note that there should be a single character in single quotes. The statement like **c = ‘gh’** will be a syntax error.

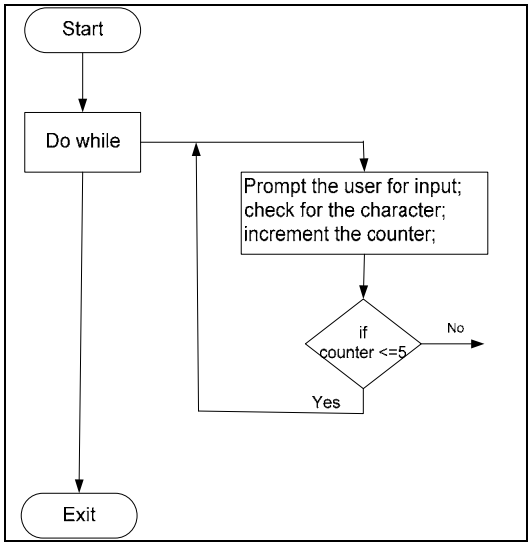
Here we use the *do-while* construct. In the *do* clause we prompt the user to enter a character.

After getting character in variable **c** from user, we compare it with our character i.e ‘z’. We use *if\else* structure for this comparison. If the character is the same as ours then we display a message to congratulate the user else we add 1 to **tryNum** variable. And then in *while* clause, we test the condition whether **tryNum** is less than or equal to 5 (*tryNum <= 5*). If this condition is true, then the body of the *do* clause is repeated again. We do this only when the condition (*tryNum <= 5*) remains true. If it is otherwise, the control goes to the first statement after the *do-while* loop.

If guess is matched in first or second try, then we should exit the loop. We know that the loop is terminated when the condition *tryNum <= 5* becomes false, so we assign a value which is greater than 5 to **tryNum** after displaying the message. Now the

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

condition in the *while* statement is checked. It proves false (as tryNum is greater than 5). So the control goes out of the loop. First look here the flow chart for the program.



The code of the program is given below.

```
//This program allows the user to guess a character from a to z
//do-while construct is used to allow five tries for guessing

# include <iostream.h>

main ( )
{
        //declare & initialize variables
int tryNum = 0 ;
        char c ;

        // do-while construct
        do
        {
        cout << "Please enter a character between a-z for guessing :     " ;
        cin >> c ;
        //check the entered character for equality
if ( c == 'z')
        {
        cout << "Congratulations,  Your guess is correct" ;
        tryNum = 6;
}
else
{
        tryNum = tryNum + 1;
}
}
        while ( tryNum <= 5);
```

vubookshoppk.com

```
}
```

There is an elegant way to exit the loop when the correct number is guessed. We change the condition in *while* statement to a compound condition. This condition will check whether the number of tries is less than or equal to 5 and the variable **c** is not equal to 'z'. So we will write the *while* clause as *while (tryNum <= 5 && c != 'z')*; Thus when a single condition in this compound condition becomes false, then the control will exit the loop. Thus we need not to assign a value greater than 5 to variable **tryNum**. Thus the code of the program will be as:

```
//This program allows the user to guess a character from a to z
//do-while construct is used to allow five tries for guessing

# include <iostream.h>

main ( )
{
        //declare & initialize variables
int tryNum = 0 ;
        char c ;

        // do-while construct, prompt the user to guess a number and compares it

        do
        {
        cout << "Please enter a character between a-z for guessing :     " ;
        cin >> c ;
        //check the entered character for equality
if ( c == 'z')
        {
        cout << "Congratulations,  Your guess is correct" ;
}
else
{
        tryNum = tryNum + 1;
}
}
        while ( tryNum <= 5 && c != 'z' );
}
```

The output of the program is given below.

```
Please enter a character between a-z for guessing :      g
Please enter a character between a-z for guessing :      z
Congratulations, Your guess is correct
```

vubookshoppk.com

for Loop

Let's see what we do in a loop. In a loop, we initialize variable(s) at first. Then we set a condition for the continuation/termination of the loop. To meet the condition to terminate the loop, we affect the condition in the body of the loop. If there is a variable in the condition, the value of that variable is changed within the body of the loop. If the value of the variable is not changed, then the condition of termination of the loop will not meet and loop will become an infinite one. So there are three things in a loop structure i.e. (i) initialization, (ii) a continuation/termination condition and (iii) changing the value of the condition variable, usually the increment of the variable value.
To implement these things, C provides a loop structure known as *for loop*. This is the most often used structure to perform repetition tasks for a known number of repetitions. The syntax of *for loop* is given below.

```
for ( initialization condition ; continuation condition ; incrementing condition )
{
        statement(s) ;
}
```

We see that a '*for statement*' consists of three parts. In initialization condition, we initialize some variable while in continuation condition, we set a condition for the continuation of the loop. In third part, we increment the value of the variable for which the termination condition is set.
Let's suppose, we have a variable **counter** of type **int.** We write *for loop* in our program as

```
for ( counter = 0 ; counter < 10 ; counter = counter +1 )
                {
cout << counter << endl;
}
```

This '*for loop*' will print on the screen 0, 1, 2 …. 9 on separate lines (as we use *endl* in our *cout* statement). In *for loop*, at first, we initialize the variable *counter* to 0. And in the termination condition, we write *counter < 10*. This means that the loop will continue till value of counter is less than 10. In other words, the loop will terminate when the value of counter is equal to or greater than 10. In the third part of *for* statement, we write *counter = counter + 1* this means that we add 1 to the existing value of *counter*. We call it incrementing the variable.

Now let's see how this loop executes. When the control goes to *for* statement first time, it sets the value of variable counter to 0, tests the condition (i.e. *counter < 10*). If it is true, then executes the body of the loop. In this case, it displays the value of *counter* which is 0 for the first execution. Then it runs the incrementing statement (i.e. counter = counter + 1). Thus the value of counter becomes 1. Now, the control goes to *for* statement and tests the condition of continuation. If it is true, then the body of the loop is again executed which displays 1 on the screen. The increment statement is again executed and control goes to *for* statement. The same tasks are repeated. When the value of counter becomes 10, the condition *counter < 10* becomes false. Then the loop is terminated and control goes out of *for* loop.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

The point to be noted is that, the increment statement (third part of *for* statement) is executed after executing the body of the loop. Thus *for* structure is equivalent to a *while* structure, in which, we write explicit statement to change (increment/decrement) the value of the condition variable after the last statement of the body. The *for* loop does this itself according to the increment statement in the *for structure.* There may be a situation where the body of *for* loop, like *while* loop, may not be executed even a single time. This may happen if the initialization value of the variable makes the condition false. The statement in the following for loop will not be executed even a single time as during first checking, the condition becomes false. So the loop terminates without executing the body of the loop.

```
for ( counter = 5 ; counter < 5 ; counter ++)
{
        cout << "The value of counter is   " << counter ;
}
```

Sample Program 1

Let's take an example to explain *for* loop. We want to write a program that prints the table of 2 on the screen.
In this program, we declare a variable **counter** of type **int**. We use this variable to multiply it by 2 with values 1 to 10. For writing the table of 2, we multiply 2 by 1, 2, 3 .. upto 10 respectively and each time display the result on screen. So we use *for* loop to perform the repeated multiplication.

Following is the code of the program that prints the table of 2.

```
//This program display the table of 2 up to multiplier 10

# include <iostream.h>

main ( )
{
        int counter;
        //the for loop
        for ( counter = 1 ; counter <= 10 ; counter = counter + 1)
        {
        cout << "2 x " << counter << " = " << 2 * counter << "\n" ;
}
}
```

This is a simple program. In the *for statement,* we initialize the variable counter to 1 as we want the multiplication of 2 starting from 1. In the condition clause, we set the condition *counter <= 10* as we want to repeat the loop for 10 times. And in the incrementing clause, we increment the variable counter by 1.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

In the body of the *for loop*, we write a single statement with *cout*. This single statement involves different tasks. The portion '<< *"2 x "'* displays the string "2 x " on the screen. After this, the next part '<< *counter'* will print the value of **counter**. The '<< " = "' will display ' = ' and then the next part '<< *2 * counter'* will display the result of 2 multiply by *counter* and the last <<"\n" (the new line character) will start a new line. Thus in the first iteration where the value of **counter** is 1, the *cout* statement will display the following line

2 x 1 = 2

After the execution of *cout* statement, the *for statement* will increment the counter variable by 1. Thus value of counter will be 2. Then condition will be checked which is still true. Thus the body of for loop (here the *cout* statement) will be executed again having the value of counter 2. So the following line will be printed.

2 x 2 = 4

The same action will be repeated 10 times with values of counter from 1 to 10. When the value of counter is 11, the condition (*counter <= 10*) will become false and the loop will terminate.

The output of the above program is as the following.

```
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
2 x 10 = 20
```

vubookshoppk.com

Now what will we do, if some one says us to write a table of 3, or 4 or 8 or any other number. Here comes the point of re-usability and that a program should be generic. We write a program in which a variable is used instead of a hard code number. We prompt the user to enter the number for which he wants a table. We store this number in the variable and then use it to write a table. So in our previous example, we now use a variable say **number** where we were using 2. We also can allow the user to enter the number of multipliers up to which he wants a table. For this, we use a variable **maxMultiplier** and execute the loop for **maxMultiplier** times by putting the condition *counter <= maxMultiplier*. Thus our program becomes generic which can display a table for any number and up to any multiplier.

Thus, the code of our program will be as below:

```
//This program takes an integer input from user and displays its table
//The table is displayed up to the multiplier entered by the user

# include <iostream.h>
```

```
main ( )
{
        int counter, number, maxMultiplier ;

        // Prompt the user for input
        cout << "Please enter the number for which you want a table :    " ;
        cin >> number ;
        cout << "Please enter the multiplier up to which you want a table :   " ;
        cin >> maxMultiplier ;

        //the for loop
        for ( counter = 1 ; counter <= maxMultiplier ; counter = counter + 1)
        {
        cout << number << " x " << counter << " = " <<  number * counter << "\n" ;
}
}
```

The output of the program is shown as follows:

```
Please enter the number for which you want a table :     7
Please enter the multiplier up to which you want a table :    8
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
```

vubookshoppk.com

Here is a guideline for programming style. We should avoid using constant values in our calculations or in long routines. The disadvantage of this is that if we want to change that constant value later, then we have to change every occurrence of that value in the program. Thus we have to do a lot of work and there may be some places in code where we do not change that value. To avoid such situations, we can use a variable at the start and assign that constant value to it and then in the program use that variable. Thus, if we need to change the constant value, we can assign the new value to that variable and the remaining code will remain the same. So in our program where we wrote the table of 2, we can use a variable (say number) and assign it the value 2. And in *cout* statement we use this variable instead of constant 2. If we want that the program should display a table of 5, then we just change the value of the variable. So for good programming, use variables for constant values instead of explicit constant values.

Increment Decrement Operators

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

We have seen that in while, do-while and for loop we write a statement to increase the value of a variable. For example, we used the statements like *counter = counter + 1;* which adds 1 to the variable *counter*. This increment statement is so common that it is used almost in every repetition structure (i.e. in while, do-while and for loop). The C language provides a unary operator that increases the value of its operator by 1. This operator is called increment operator and sign ++ is used for this. The statement *counter = counter + 1;* can be replaced with the statement
counter ++ ;
The statement *counter*++ adds 1 to the variable **counter**. Similarly the expressions i = i + 1 ; and j = j + 1 ; are equivalent to **i++** ; and **j++**; respectively. There is also an operator **--** called decrement operator. This operator decrements, the value of its operand by 1. So the statements *counter = counter - 1;* and *j = j - 1;* are equivalent to **counter--;** and **j--;** respectively.

The increment operator is further categorized as pre-increment and post-increment. Similarly, the decrement operator, as pre-decrement and post-decrement.

In pre-increment, we write the sign before the operand like **++j** while in post-increment, the sign ++ is used after the operand like **j++**. If we are using only variable increment, pre or post increment does not matter. In this case, **j++** is equivalent to **++j**. The difference of pre and post increment matters when the variable is used in an expression where it is evaluated to assign a value to another variable. If we use pre-increment (++j), the value of **j** is first increased by 1. This new value is used in the expression. If we use post increment (j++),the value of j is used in the expression. After that it is increased by 1. Same is the case in pre and post decrement.

If j = 5, and we write the expression
x = ++ j ;
After the evaluation of this expression, the value of x will be 6 (as j is incremented first and then is assigned to x). The value of **j** will also be 6 as ++ operator increments it by 1.

If j = 5, and we write the expression
x = j++ ;
Then after the evaluation of the expression, the value of x will be 5 (as the value of j is used before increment) and the value of j will be 6.
The same phenomenon is true for the decrement operator with the difference that it decreases the value by 1. The increment and decrement operators affect the variable and update it to the new incremented or decremented value.

The operators ++ and -- are used to increment or decrement the variable by 1. There may be cases when we are incrementing or decrementing the value of a variable by a number other than 1. For example, we write *counter = counter + 5*; or *j = j – 4*;. Such assignments are very common in loops, so C provides operators to perform this task in short. These operators do two things they perform an action (addition, subtraction etc) and do some assignment.
These operators are +=, -=, *=, /= and %=. These operators are compound assignment operators. These operators assign a value to the left hand variable after performing an

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

action (i.e. +, -, *, / and %). The use of these operators is explained by the following examples.

Let's say we have an expression, *counter = counter + 5;*. The equivalent of this expression is *counter += 5*;. The statement *counter += 5*; does two tasks. At first, it adds 5 to the value of **counter** and then assigns this result to **counter**. Similarly the following expressions

```
x = x + 4 ;
x = x - 3 ;
x = x * 2 ;
x  = x  / 2 ;
x  = x  % 3;
```

can be written in equivalent short statements using the operators (+=, -=, *=, /=, %=) as follows

```
x  += 4 ;
x  -= 3 ;
x  *= 2;
x  /= 2;
x  %= 3 ;
```

Note that there is no space between these operators. These are treated as single signs. Be careful about the operator %=. This operator assigns the remainder to the variable. These operators are alternate in short hand for an assignment statement. The use of these operators is not necessary. A programmer may use these or not. It is a matter of style.

Example Program 2

Let's write a program using *for* loop to find the sum of the squares of the integers from 1 to n. Where n is a positive value entered by the user (i.e. Sum = $1^2 + 2^2 + 3^2 +$ ……$+ n^2$)

The code of the program is given below:

```
//This program displays the sum of squares of integers from 1 to n

# include <iostream.h>

main ( )
{
        //declare and initialize variables
int i, n, sum;
        sum = 0 ;

        //get input from user and construct a for loop
cout << "Please enter a positive number for sum of squares:    " ;
        cin >> n;
```

vubookshoppk.com

```
        for ( i = 1 ; i <= n ; i ++)
        {
        sum += i * i ;
}
cout << "The sum of the first " << n << " squares is  " << sum << endl ;
}
```

In the program declared three variables **i**, **n** and **sum**. We prompted the user to enter a positive number. We stored this number in the variable **n**. Then we wrote a *for* loop. In the initialization part, we initialized variable **i** with value 1 to start the counting from 1. In the condition statement we set the condition **i** less than or equal to **n** (number entered by the user) as we want to execute the loop **n** times. In the increment statement, we incremented the counter variable by 1. In the body of the **for** loop we wrote a single statement *sum += i * i* ;. This statement takes the square of the counter variable (i)and adds it to the variable **sum**. This statement is equivalent to the statement *sum = sum + (i * i)* ; Thus in each iteration the square of the counter variable (which is increased by 1 in each iteration) is added to the **sum**. Thus loop runs **n** times and the squares of numbers from 1 to **n** are summed up. After completing the *for* loop the *cout* statement is executed which displays the sum of the squares of number from 1 to **n.**

Following is the output when the number 5 is entered.

```
Please enter a positive number for sum of squares:      5
The sum of the first 5 squares is 55
```

Tips

Comments should be meaningful, explaining the task
Don't forget to affect the value of loop variable in *while* and *do-while* loops
Make sure that the loop is not an infinite loop
Don't affect the value of loop variable in the body of *for* loop, the *for* loop does this by itself in the *for statement*
Use pre and post increment/decrement operators cautiously in expressions

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Lecture No. 8

Reading Material

Deitel & Deitel – C++ How to Program Chapter 2

2.16, 2.18

Summary

- Switch Statement
- Break Statement
- Continue Statement
- Guide Lines
- Rules for structured Programming/Flow Charting
- Sample Program
- Tips

Switch Statement

Sometimes, we have multiple conditions and take some action according to each condition. For example, in the payroll of a company, there are many conditions to deduct tax from the salary of an employee. If the salary is less than Rs. 10000, there is no deduction. But if it falls in the slab Rs. 10000 - 20000, then the income tax is deducted. If it exceeds the limit of Rs. 20000, some additional tax will be deducted. So the appropriate deduction is made according to the category or slab of the salary. We can also understand this from the example of grades secured by the students of a class. Suppose we want to print description of the grade of a student. If the student has grade 'A' we print 'Excellent' and 'Very good', 'good', 'poor' and 'fail' for grades B, C, D, and F respectively. Now we have to see how this multi-condition situation can be applied in a program. We have a tool for decision making i.e. '*if statement'*. We can use '*if statement'* to decide what description for a grade should be displayed. So we check the grade in *if statement* and display the appropriate description. We have five categories of grades-- A, B, C, D, and F. We have to write five *if statements* to check all the five possibilities (probabilities) of grade. So we write this in our program as under-

```
if ( grade == 'A' )
   cout << "Excellent" ;
if ( grade == 'B' )
   cout << "Very Good" ;
if ( grade == 'C' )
   cout << "Good" ;
if ( grade == 'D' )
   cout << "Poor" ;
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
if ( grade == 'F' )
   cout << "Fail" ;
```

These statements are correct and perform the required task. But the '*if statement*' is computationally one of the most expensive statements in a program. We call it expensive due to the fact that the processor has to go through many cycles to execute an *if statement* to evaluate a single decision. So to make a program more efficient, try to use the minimum number of *if statements*. This will make the performance of the program better.
So if we have different conditions in which only one will be true as seen in the example of student's grades, the use of *if statement* is very expensive. To avoid this expensiveness, an alternate of multiple *if statements* can be used that is *if/else statements*. We can write an *if statement* in the body of an *if statement which is* known as *nested if*. We can write the previous code of *if statements* in the following *nested if/else* form.

```
If ( grade == 'A' )
     cout << "Excellent" ;
else if ( grade == 'B' )
     cout << "Very Good" ;
else if ( grade == 'C' )
     cout << "Good" ;
else if ( grade == 'D' )
     cout << "Poor" ;
else if ( grade == 'F' )
     cout << "Fail" ;
```

In the code, there is single statement with each *if statement*. If there are more statements with an *if statement*, then don't forget the use of braces and make sure that they match (i.e. there is a corresponding closing brace for an opening brace). Proper indentation of the blocks should also be made.
In the above example, we see that there are two approaches for a multi way decision. In the first approach, we use as many *if statements* as needed. This is an expensive approach. The second is the use of nested *if statements*. The second is little more efficient than the first one. In the '*nested if statements*' the nested else is not executed if the first *if* condition is true and the control goes out of the *if* block.
The C language provides us a stand-alone construct to handle these instances. This construct is *switch structure*. The *switch structure* is a multiple-selection construct that is used in such cases (multi way decisions) to make the code more efficient and easy to read and understand.
The syntax of switch statement is as follows.

```
switch ( variable/expression )
{
        case constant1 : statementLlist1 ;
        case constant2 : statementLlist2 ;
                                  :
 :
case constantN : statementListN ;
        default : statementList ;
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

}

In the *switch* statement, there should be an integer variable (also include *char*) or an expression which must evaluate an integer type (whole numbers only, the decimal numbers 2.5, 14.3 etc are not allowed). We can't use compound conditions (i.e. the conditions that use logical operators && or ||) in *switch* statement and in *case* statements. The *constants* also must be integer constants (which include *char*). We can't use a variable name with the case key word. The *default* statement is optional. If there is no *case* which matches the value of the *switch* statement, then the statements of *default* are executed.
The *switch* statement takes the value of the *variable, if* there is an *expression* then it evaluates the *expression* and after that looks for its value among the *case constants*. If the value is found among the *constants* listed in *cases*, the statements in that *statementList* are executed. Otherwise, it does nothing. However if there is a default (which is optional), the statements of *default* are executed.

Thus our previous grade example will be written in switch statement as below.

```
switch ( grade )
{
        case 'A' : cout << "Excellent" ;
        case 'B' : cout << "Very Good" ;
        case 'C' : cout << "Good" ;
        case 'D' : cout << "Poor" ;
        case 'F' : cout << "Fail" ;
}
```

We know that C language is 'case sensitive'. In this language, 'A' is different from 'a'. Every character has a numeric value which is stored by the computer.. The numeric value of a character is known as ASCII code of the character. The ASCII code of small letters (a, b, c etc) are different from ASCII code of capital letters (A, B, C etc). We can use characters in switch statement as the characters are represented as whole numbers inside the computers.
Now we will see how the use of ' the letter a' instead of 'A' can affect our program. We want our program to be user- friendly. We don't want to restrict the user to enter the grade in capital letters only. So we have to handle both small and capital letters in our program. Here comes the limitations of *switch* statement. We can't say in our statement like
case 'A' or 'a' : statements ;
We have to make two separate cases so we write

```
        case 'A" :
        case 'a' :
                statements;
```

In the *switch* statement, the cases fall through the *case* which is true. All the statements after that *case* will be executed right down to the end of the *switch* statement. This is very important to understand it. Let's suppose that the user enters grade 'B'. Now the case 'A' is skipped. Next case 'B' matches and statement *cout << "Very Good" ;* is executed. After that, all the statements will be executed. So *cout << "Good"* ; *cout << "Poor"* ;and *cout << "Fail"* ; will be executed after one another.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

We don't want this to happen. We want that when a case matches, then after executing its statement, the control should jump out of the *switch* statement leaving the other cases. For this purpose we use a key word **break**.

Break Statement

The *break* statement interrupts the flow of control. We have seen in *switch* statement that when a true case is found, the flow of control goes through every statement down ward. We want that only the statements of true case should be executed and the remaining should be skipped. For this purpose, we use the *break* statement. We write the break statement after the statements of a case. Thus, when a true case is found and its statements are executed then the break statement interrupts the flow of control and the control jumps out of the *switch* statement. If we want to do the same task for two cases, like in previous example for 'A' and 'a', then we don't put *break* statement after the first case. We write both the cases (or the cases may be more than two) line by line then write the common statements to be executed for these cases. We write the *break* statement after these common statements. We should use the *break* statement necessarily after the statements of each case. The break statement is necessary in *switch* structure, without it the *switch* structure becomes illogic. As without it all the statement will execute after first match case is found.

The above code does nothing if the grade is other than these five categories (i.e. A, B, C, D and F). To handle all the possibilities of grade input, we write a default statement after the last case. The statement in this default case is executed if no case matches the grade. So in our program, we can write the *default* statement after the last case as under.

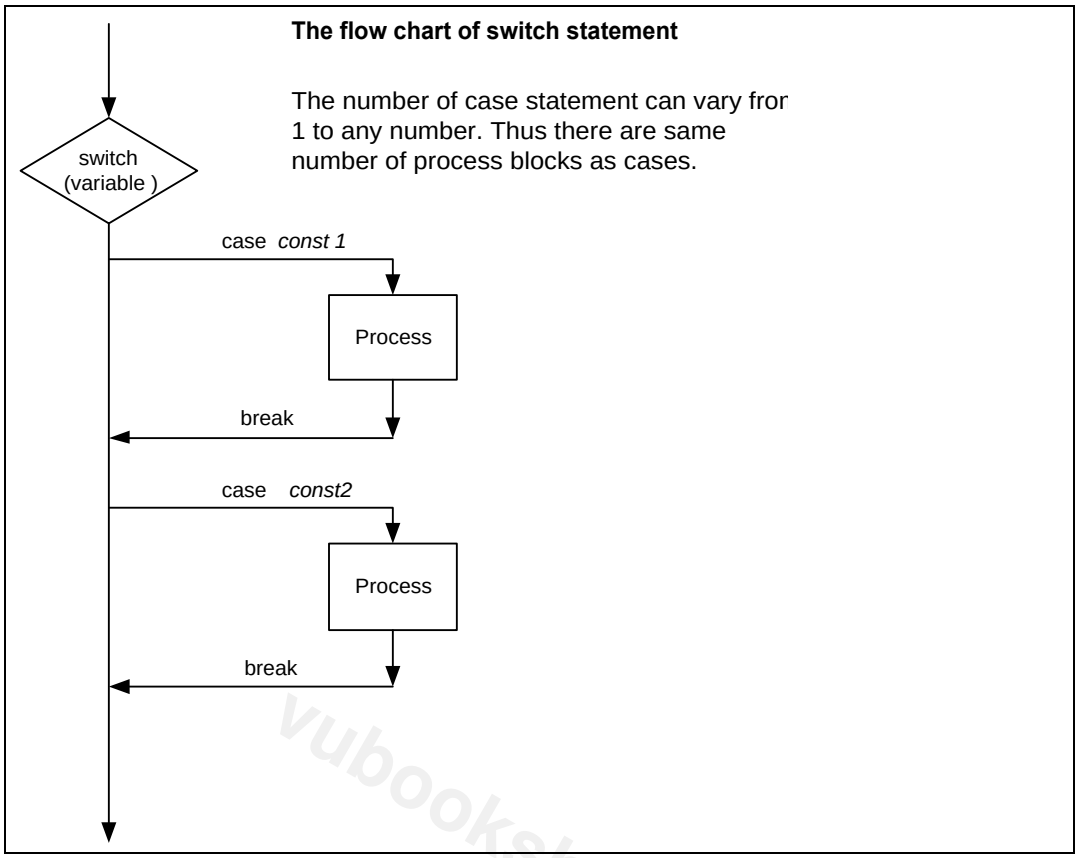
```
default : cout << "Please enter grade from A to D or F " ;
```

The break statement is also used in decision structures other than switch structure. We have seen that in *while, do-while* and *for* loops, we have to violate some condition explicitly to terminate the loop before its complete repetitions. As in a program of guessing a character, we make a variable **tryNum** greater than 5 to violate the *while* condition and exit the loop if the correct character is guessed before five tries. In these loops, we can use the *break* statement to exit a loop. When a *break* statement is encountered in a loop, the loop terminates immediately. The control exits the inner most loop if there are nested loops. The control passes to the statement after the loop. In the guessing character example, we want that if the character is guessed in first or second attempt,. then we print the message 'Congratulations, You guess is correct' and exit the loop. We can do this by using a *break* statement with an *if* statement. If the character is guessed, we print the message. Afterwards, the *break* statement is executed and the loop terminates. So we can write this as follows.

```
if ( c == 'z' )  // c is input from user
{
        cout << "Great, Your guess is correct" ;
        break;
}
```

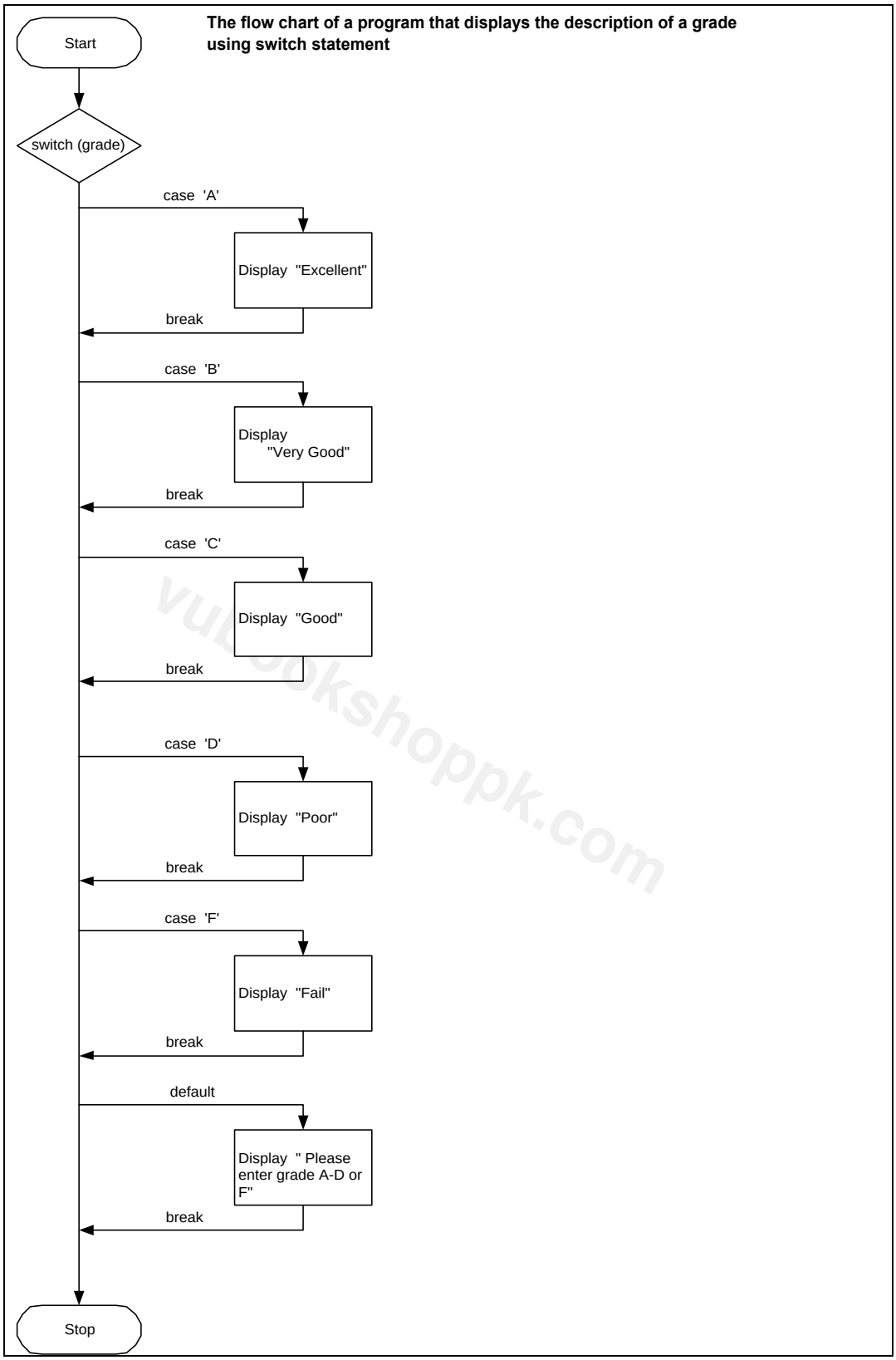
Thus, *break* statement can be used to jump out of a loop very quickly.
The flow chart of the *switch* statement is similar to *if* statement and is given below.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533



Now we can write the complete code for the program that prints the description of the grade entered by the user.
The flow chart of the program is given below.

vubookshoppk.com



The code of the program is given below.

```
//This program gets a grade from user and displays a description accordingly
```

vubookshoppk.com

```
# include <iostream.h>
main ( )
{
        char grade ;
        cout << "Please enter the student's grade :    " ;
        cin >> grade ;
        switch ( grade )
    {
                case 'A' :      // grade was upper case A
                case 'a' :      // grade was lower case a
        cout << "Excellent" ;
        break :         // necessary to exit switch
                case 'B' :      // grade was upper case B
                case 'b' :      // grade was lower case b
        cout << "Very Good" ;
        break : // necessary to exit switch
                case 'C' :      // grade was upper case C
case 'c' :      // grade was lower case c
        cout << "Good" ;
        break : // necessary to exit switch
                case 'D' :      // grade was upper case D
case 'd' :      // grade was lower case d
        cout << "Poor" ;
        break : // necessary to exit switch
                case 'F' :      // grade was upper case F
case 'f' :              // grade was lower case f
cout << "Fail" ;
break : // necessary to exit switch
default :
cout << "Please enter grade from A to D or F " ;
}
}
```

A sample out put of the program is shown here.

```
Please enter the student's grade :    b
Very Good
```

continue Statement

There is another statement relating to loops. This is the *continue* statement. Sometimes we have a lot of code in the body of a loop. The early part of this code is common that is to be executed every time (i.e. in every iteration of loop) and the remaining portion is to be executed in certain cases and may not be executed in other cases. But the loop should be continuous. For this purpose, we use the *continue* statement. Like the *break* statement, the *continue* statement is written in a single line. We write it as

```
continue ;
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

The *continue* forces the immediate next iteration of the loop. So the statements of the loop body after *continue* are not executed. The loop starts from the next iteration when a *continue* statement is encountered in the body of a loop. One can witness very subtle things while using *continue*.
Consider the *while* loop. In *while* loop, we change the value of the variable of *while* condition so that it could make the condition false to exit the loop. Otherwise, the loop will become an infinite one. We should be very careful about the logic of the program while using *continue* in a loop. Before the *continue* statement, it is necessary to change (increment/decrement) the value of the variable on which the *while* condition depends. Similarly it is same with the do-while loop. Be careful to increment or decrement the conditional variable before the *continue* statement.

In *for* loop, there is a difference. In a *while* loop when *continue* is encountered, the control goes to the *while* statement and the condition is checked. If condition is true the loop is executed again else the loop exits. In a *for* loop, the three things i.e. initialization, condition and increment/decrement are enclosed together as we write *for (counter = 0 ; counter <= 5 ; counter ++)* . In the *for* loop when a *continue* is encountered, the counter (i.e. loop variable) is incremented at first before the execution of the loop condition. Thus, in '*for* loop' the increment to the loop variable is built in and after *continue* the next iteration of the loop is executed by incrementing the loop variable. The condition is checked with the incremented value of the loop variable. In *while* and *do-while* loop, it is our responsibility to increment the value of the loop variable to test the condition. In a *for* loop, the *continue* automatically forces this increment of value before going to check the condition.

goto Statement

Up to now we have covered the basic programming constructs. These include sequences, decisions and repetition structures (i.e. loops). In sequences, we use the simple statements in a sequence i.e. one after the other. In decisions construct we use the *if statement, if/else statement*, the multi way decision construct (i.e. the *switch statement)*. And in repetition structures, we use the *while, do-while* and *for* loops. Sometime ago, two computer scientists Gome and Jacopi proved that any program can be written with the help of these three constructs (i.e. sequences, decisions and loops).
There is a statement in the computer languages COBOL, FORTRON and C. This statement is *goto* statement. The *goto* is an unconditional branch of execution. The *goto* statement is used to jump the control anywhere (back and forth) in a program. In legacy programming, the programs written in COBOL and FORTRAN languages have many unconditional branches of execution. To understand and decode such programs that contain unconditional branches is almost impossible. In such programs, it is very difficult, for a programmer, to keep the track of execution as the control jumps from one place to the other and from there to anywhere else. We call this kind of traditional code as spagatti code. It is very difficult to trace out the way of execution and figure out what the program is doing. And debugging and modifying such programs is very difficult.

When structured programming was started, it was urged not to use the *goto* statement. Though *goto* is there in C language but we will not use it in our programs. We will

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

adopt the structured approach. All of our programs will consist of sequences, decisions and loops.

Guide Lines

In general, we should minimize the use of break statement in loops. The switch statement is an exception in this regard where it is necessary to use the *break* statement after every case. Otherwise, there may be a logical error. While writing loops, we should try to execute the loops with the condition test and should try to avoid the *break* statement. The same applies to the *continue* statement. The *continue* statement executes some statements of the loop and then exits the loop without executing some statements after it. We can use the *if* statement for this purpose instead of *continue*. So never use the *goto* statement and minimize the usage of *break* and *continue* statements in loops. This will make the code easy to understand for you and for others. Moreover the additions and modifications to such code will be easy, as the path of execution will be easy to trace.
Make a program modular. This means that divide a large program into small parts. It will be easy to manage these small parts rather than a larger program. There should be single entry and single exit in every module or construct. The use of *break* statement in a construct violates this rule as a loop having a *break* statement can exit through *break* statement or can terminate when the loop condition violates. As there are two exit points, this should be avoided. The single entry- single exit approach makes the execution flow simple.
Here is an example from daily life, which shows that single entry and single exit makes things easy. You would have often seen at a bus stop, especially in rush hours, that when a bus reaches the stop, everyone tries to jump into the bus without caring for others. The passengers inside the bus try to get down from the vehicle. So you see there a wrestling like situation at the door of the bus. Separate doors for entering or exiting the bus can be the solution. In this way, the passengers will easily enter or exit the bus.
We have applied this single entry and single exit rule in drawing our flow charts. In the flow charts, we draw a vertical line from top to down. The point where the line starts is our entry point and downward at the same line at the end is our exit point. Our all other processes and loops are along or within these two points. Thus our flow charts resemble with the code.

Rules for Structured Programming/Flow Charting

There are few simple rules for drawing structured flow charts of programs. One should be familiar with these.
Rule No:1-Start with the simple flow chart. This means that draw a start symbol, draw a rectangle and write in it whatsoever you want to do and then draw a stop symbol. This is the simplest flow chart.
Rule No:2- Any rectangle (a rectangle represents a process which could be input, output or any other process) can be replaced by two rectangles.
This concept is the same as taking a complex problem and splitting it up into two simpler problems. So we have 'split it up' method to move towards a modular

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

approach. So start with a block (rectangle) and then any rectangle can be replaced by two rectangles (blocks).

Rule No:3- Any rectangle can be replaced with a structured flow charting construct. These construct include decisions, loops or multi- way decision. This means that we can put a structure of an *if* construct or *switch* construct in the place of a rectangle. Here we come to know the advantage of single entry and single exit concept. This single entry and single exit block can be replaced with a rectangle.

Rule No: 4- This rule states that rule number 2 and 3 can be repeated as many times as you want.

By using these rules we are splitting a problem into simpler units so that each part can be handled either by sequences (one rectangle, second rectangle and so on) or by a decision (if, if/else, switch or by a loop). Through this approach, a large problem can be solved easily.

The flow charts drawn with these rules and indented to the left side will have one to one correspondence with our code. Thus it becomes very easy to identify the code that is written for a specific part of the flow chart. In this way the code can easily be debugged.

Sample Program

Let's consider a problem. In a company, there are deductions from the salary of the employees for a fund. The deductions rules are as follows:

If salary is less than 10,000 then no deduction

If salary is more than 10,000 and less than 20,000 then deduct Rs. 1,000 as fund

If salary is equal to or more than 20,000 then deduct 7 % of the salary for fund

Take salary input from user and after appropriate deduction show the net payable amount.

Solution

As we see that there is multi way decision in this problem, so we use *switch* statement. The salary is the switch variable upon which the different decisions depend. We can use only a single constant in *case* statement. So we divide the salary by 10000 to convert it into a single case constant. As we know that in integer division we get the whole number as the answer. Thus if answer is 0 the salary is less than 10000, if answer is 1 then it is in range 10000 to 19999 (as any amount between 10000 – 19999 divided by 10000 will result 1). If the answer is greater than 1, it means the salary is equal to or more than 20000.

Following is the complete code of our program.

```
// This program gets salary input from user and calculates and displays the net payable
// amount after deduction according the conditions

# include <iostream.h>
main ( )
{
        int salary ;
float deduction, netPayable ;
        cout << "Please enter the salary :    " ;
        cin >> salary ;
        // here begins the switch statement
```

```
        switch ( salary / 10000 ) // this will produce a single value
        {
        case 0 :          // this means salary is less than 10,000
                deduction = 0; // as deduction is zero in this case
netPayable = salary ;
                cout << "Net Payable (salary – deduction)  =  " ;
cout << salary << " - " << deduction << " = " << netPayable;
                break;          //necessary to exit switch
case 1 :        // this means salary is in range 10,000 – 19,999
                deduction =  1000 ;
                netPayable = salary – deduction ;
                cout << "Net Payable (salary – deduction)  =  " ;
cout << salary << " - " << deduction << " = " << netPayable;
                break;          //necessary to exit switch
        default :               // this means the salary is 20,000 or more
                deduction =  salary * 7 /100 ;
                netPayable = salary – deduction ;
                cout << "Net Payable (salary – deduction)  =  " ;
cout << salary << " - " << deduction << " = " << netPayable;
}
}
```

Here is the out put of the program.

```
Please enter the salary :    15000
Net Payable (salary – deduction) = 15000 – 1000 = 14000
```

Tips

Try to use the *switch* statement instead of multiple *if* statements
Missing a *break* statement in a *switch* statement may cause a logical error
Always provide a *default* case in switch statements
Never use *goto* statement in your programs
Minimize the use of *break* and *continue* statements

vubookshoppk.com

Lecture No. 9

Reading Material

Deitel & Deitel – C++ How to Program — chapter 2, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6

Summary

- Introduction
- Functions
- Structure of a Function
- Declaration and Definition of a Function
- Sample Program 1
- Sample Program 2
- Sample Program 3
- Summary
- Tips

Introduction

Now our toolkit is almost complete. The basic constructs of programming are sequence, decision making and loops. You have learnt all these techniques. Now we can write almost all kinds of programs. There are more techniques to further refine the programs. One of the major programming constructs is Functions. C is a function-oriented language. Every program is written in different functions.

In our daily life, we divide our tasks into sub tasks. Consider the making of a laboratory stool.

It has a seat and three legs. Now we need to make a seat and three legs out of wood. The major task is to make a stool. Sub tasks are, make a seat and then fabricate three legs. The legs should be identical. We can fashion one leg and then re-using this prototype, we have to build two more identical legs. The last task is to assemble all these to make a stool. We have a slightly difficult task and have broken down it into simpler pieces. This is the concept of functional design or top-down designing. In top design, we look at the problem from top i.e. identification of the problem. What we

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

have to solve? Then refine it and divide it into smaller pieces. We refine it again and divide it into smaller pieces. We keep on doing it as long as we get easily manageable task. Let's consider an example like home construction. From the top level, we have to construct a home. Then we say that we need design of the home according to which the building will be constructed. We need to construct rooms. How can we construct a room? We need bricks, cement, doors, windows etc. Procurement of all of these things is tasks. Once we come down to the level where a task is easily manageable and doable, we stop doing further refinement. When we break up a task into smaller sub tasks, we stop at a reasonable level. Top-down designing mechanism is based on the principle of 'divide and conquer' i.e. we divide a big task into smaller tasks and then accomplish them.

Let's have a look at a simple example to understand the process of dividing big task into simple ones. Suppose we want to know how many students are currently logged in the LMS (Learning Management System) of VU. This task will be handed over to the network administrator to find out the number of students currently logged in LMS of the university. The network administrator will check the network activity or get this information from the database and get the list of students currently logged in. The number of students is counted from that list and the result is given back to us. What has happened in this whole process? There was a simple request to find the number of students currently logged in LMS. This request is delegated to the network administrator. The network administrator performs this task and we get the result. In the mean time, we can do some other task as we are not interested in the names or list of students. We only want the number of students. This technique is known as parallel processing. In terms of programming, network administrator has performed a function i.e. calculation of the number of students. During this process, the network administrator also gets the list of students which is hidden from us. So the information hiding is also a part of the function. Some information is given to the network administrator (i.e. the request to calculate the number of students currently logged in the LMS) while some information is provided back to us (i.e. the number of students).

Functions

The functions are like subtasks. They receive some information, do some process and provide a result. Functions are invoked through a calling program. Calling program does not need to know what the function is doing and how it is performing its task. There is a specific function-calling methodology. The calling program calls a function by giving it some information and receives the result.

We have a main ( ) in every C program. ‘main ( )’ is also a function. When we write a function, it must start with a name, parentheses, and surrounding braces just like with main ( ). Functions are very important in code reusing.

There are two categories of functions:

1. Functions that return a value
2. Functions that do not return a value

Suppose, we have a function that calculates the square of an integer such that function will return the square of the integer. Similarly we may have a function which displays

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

some information on the screen so this function is not supposed to return any value to the calling program.

Structure of a Function

The declaration syntax of a function is as follows:

```
return-value-type    function-name( argument-list )
{
     declarations and statements
}
```

The first line is the function header and the declaration and statement part is the body of the function.

return-value_type:
Function may or may not return a value. If a function returns a value, that must be of a valid data type. This can only be one data type that means if a function returns an int data type than it can only return int and not char or float. Return type may be int, float, char or any other valid data type. How can we return some value from a function? The keyword is **return** which is used to return some value from the function. It does two things, returns some value to the calling program and also exits from the function. We can only return a value (a variable or an expression which evaluates to some value) from a function. The data type of the returning variable should match *return_value_type* data type.

There may be some functions which do not return any value. For such functions, the *return_value_type* is **void**. 'void' is a keyword of 'C' language. The default *return_value_type* is of *int* data type i.e. if we do not mention any *return_value_type* with a function, it will return an *int* value.

Function-name:
The same rules of variable naming conventions are applied to functions name. Function name should be self-explanatory like square, squareRoot, circleArea etc.

argument-list:
Argument list contains the information which we pass to the function. Some function does not need any information to perform the task. In this case, the argument list for such functions will be empty. Arguments to a function are of valid data type like int number, double radius etc.

Declarations and Statements:
This is the body of the function. It consists of declarations and statements. The task of the function is performed in the body of the function.

Example:

```
//This function calculates the square of a number and returns it.
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
int square(int number)
{
        int result = 0;
        result = number * number;
        return result;
}
```

Calling Mechanism:

How a program can use a function? It is very simple. The calling program just needs to write the function name and provide its arguments (without data types). It is important to note that while calling a function, we don't write the return value data type or the data types of arguments.

Example:

```
//This program calculates the square of a given number

#include <iostream.h>

        main()
        {
                int number, result;
                result = 0;
                number = 0;
                // Getting the  input from the user
                cout << " Please enter the number to calculate the square ";
                cin >> number;

                // Calling the function square(int number)
                result = square(number);
                cout << " The square of " << number << " is " << result;
        }
```

vubookshoppk.com

Declaration and Definition of a Function

Declaration and definition are two different things. Declaration is the prototype of the function, that includes the return type, name and argument list to the function and definition is the actual function code. Declaration of a function is also known as signature of a function.
As we declare a variable like *int x;* before using it in our program, similarly we need to declare function before using it. Declaration and definition of a function can be combined together if we write the complete function before the calling functions. Then we don't need to declare it explicitly. If we have written all of our functions in a different file and we call these functions from *main( )* which is written in a different file. In this case, the *main( )* will not be compiled unless it knows about the functions

declaration. Therefore we write the declaration of functions before the *main( )* function. Function declaration is a one line statement in which we write the return type, name of the function and the data type of arguments. Name of the arguments is not necessary. The definition of the function contains the complete code of the function. It starts with the declaration statement with the addition that in definition, we do write the names of the arguments. After this, we write an opening brace and then all the statements, followed by a closing brace.

Example:

If the function square is defined in a separate file or after the calling function, then we need to declare it:

Declaration:

```
int square ( int );
```

Definition:

```
int square ( int number)
{
        return (number * number ) ;
}
```

vubookshoppk.com

Here is the complete code of the program:

```
//This program calculates the square of a given number

#include <iostream.h>

        // Function declarations.
        int square(int);

        main()
        {
                int number, result;
                result = 0;
                number = 0;
                cout << " Please enter the number to calculate the square ";
                cin >> number;
                // Calling the function square(int number)
                result = square(number);
                cout << " The square of " << number << " is " << result;
        }
```

```
// function to calculate the square of a number
int square ( int number)
{
        return (number * number ) ;
}
```

A function in a calling program can take place as a stand-alone statement, on right-hand side of a statement. This can be a part of an assignment expression.

Considering the above example, here are some more ways of function calling mechanism.

```
result = 10 + square (5);
                or
        result = square (number + 10);
                or
        result = square (number) + square (number + 1) + square (3 * number);
                or
        cout << " The square of  " << number << " is " << square (number);
```

In the above statements, we see that functions are used in assignment statements. In a statement *result = square(5);* The *square(5)* function is called and the value which is returned from that function (i.e. the value returned within the function using the *return* keyword) is assigned to the variable *result*. In this case, the *square(5)* will return 25, which will be assigned to variable *result*. There may be functions which do not return any value. These functions can't be used in assignment statements. These functions are written as stand-alone statements.

vubookshoppk.com

Sample Program 1

C is called function-oriented language. It is a very small language but there are lots of functions in it. Function can be on a single line, a page or as complex as we want.

Problem statement:
Calculate the integer power of some number (x^n).

Solution:
We want to get the power of some number. There is no operator for power function in C. We need to write a function to calculate the power of x to n (i.e. x^n). How can we calculate the power of some number? To get the power of some number x to n, we need to multiply x with x up to n times. Now what will be the input (arguments) to the function? A number and power, as number can be a real number so we have to declare number as a double date type and the power is an integer value so we will declare the power as an integer. The power is an integer value so we will declare power as an integer. The result will also be a real number so the return value type will be of double data type. The function name should be descriptive, we can name this function as *raiseToPow*. The declaration of the function is:

```
double raiseToPow ( double x,  int power ) ;
```

To calculate the power of *x* up to *power* times, we need a loop which will be executed *power* times. The definition of function is:

```
// function to calculate the power of some number

double raiseToPow ( double x , int power )
        {
                double result ;
                int i ;
                result = 1.0 ;

                for ( i = 1 ; i <= power ; i ++ )
                {
                        result *= x ; // same as result  = result  * x
                }
                return ( result ) ;
        }
```

Here is the program which is calling the above function.

```
// This program is calling a function raiseToPow.

#include <iostream.h>

//Function declaration
double raiseToPow ( double , int )


main ( )
{
        double x ;
        int i ;
        cout << " Please enter the number " ;
        cin >> x ;
        cout << " Please enter the integer power that you want this  number raised to " ;
        cin >> i ;
        cout << x << " raise to power  " << i << " is equal to " << raiseToPow ( x , i ) ;
}
```

Now we have to consider what will happen to the values of arguments that are passed to the function? As in the above program, we are passing **x** and **i** to the *raiseToPow* function. Actually nothing is happening to the values of **x** and **i**. These values are unchanged. A copy of values x and i are passed to the function and the values in the calling program are unchanged. Such function calls are known as 'call by value'.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

There is another way to call a function in which the function can change the values of variables that are passed as arguments, of calling program. Such function call is known as call by reference.

Sample Program 2

Problem statement:
Calculate the area of a ring.

Solution:
We know that a ring consists of a small circle and a big circle. To calculate the area of a ring, we have to subtract the area of small circle from the area of big circle. Area of any circle is calculated as Pi * r^2. We write a function to calculate the area of a circle and use this function to calculate the area of small circle and big circle.

Following is the code of the function circleArea:

```
// Definition of the circleArea function.

double circleArea ( double radius )
{
        // the value of Pi = 3.1415926
        return ( 3.1415926 * radius * radius ) ;
}
```

Here is the complete code of the calling program.

```
// This program calculates the area of a ring

#include <iostream.h>

// function declaration.
double circleArea ( double);

void main ( )
{
        double rad1 ;
        double rad2 ;
        double ringArea ;

        cout << " Please enter the outer radius value: " ;
        cin >> rad1 ;
        cout << " Please enter the radius of the inner circle: " ;
        cin >> rad2 ;

        ringArea = circleArea ( rad1 ) – circleArea (rad2 ) ;
```

vubookshoppk.com

```
        cout<< " Area of the ring having inner raduis " << rad2 << " and the outer radius " <<
                rad1 << " is " << ringArea ;
}

double circleArea ( double radius )
{
        // the value of Pi = 3.1415926
        return ( 3.1415926 * radius * radius ) ;
}
```

Sample Program 3

There are some other kinds of functions which are used to test some condition. Such functions return *true* or *false*. These functions are very important and used a lot in programming. In C condition statements, the value zero (0) is considered as false and any value other than zero is considered as true. So the return type of such functions is *int*. We usually return 1 when we want the function to return true and return 0 when we want the function to return 0. Here is a sample program to elaborate this.

vubookshoppk.com

Problem statement:
Write a function which tests that a given number is even or not? It should return true if the number is even, otherwise return false.

Solution:
We already know the method of deciding whether a number is even or not. The name of the function is *isEven*. Its return type will be *int*. It will take an *int* as an argument. So the declaration of the function should be as below;

```
int isEven ( int ) ;
```

We can also use a function in the conditional statements like:

```
if ( isEven ( number ) )
```

If the number is even, the function will return none zero value (i.e. usually 1) and the *if statement* will be evaluated as true. However, if the number is odd, the function will return a zero value and the *if statement* is evaluated as false.

Here is a complete program.

```
// This program is calling a function to test the given number is even or not

#include <iostream.h>

// function declaration.
int isEven(int);

void main ( )
```

```
{
        int number;

        cout << " Please enter the number: " ;
        cin >> number ;

        if ( isEven ( number ) )
        {
                cout << " The number entered is even " << endl;
        }
        else
        {
                cout << " The number entered is odd " << endl;
        }
}

int isEven ( int number )
{
        if ( 2 * ( number / 2 ) == number )
        {
                return 1;
        }
        else
        {
                return 0;
        }
}
```

vubookshoppk.com

Summary

Functions are very good tools for code reuse. We have seen in the above example that the area of two circles has been calculated without rewriting the code. This means that the code has been reused. We can reuse the *circleArea* function to find the area of any circle.
A function performs a specific task. Functions also provide encapsulation. The calling program does not know how the function is performing its task. So we can build up modular form from small building blocks and build up more and more complex programs.

If we are going to use a function in our program and the definition of the function is after the calling program. The calling program needs to know how to call the function, what the arguments are and what it will return. So its declaration must occur before usage. If we do not declare a function before using, the compiler will give an error. If we define a function before the calling program, then we do not need a separate declaration. The function declaration is also known as function prototype or function signature. Whenever, we need to build something, first of all we build a prototype of that thing and then later on we build it. Similarly the function declaration is used as a

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

prototype. We are following the top- down methodology. We break the program into smaller modules and just declare the functions and later on we can define these.

Exercise:

1. Modify the *raise to power* function so that it can handle negative power of x, zero and positive power of x.

2. Modify the *area of ring* function put in error checking mechanism.

Tips

- We used functions for breaking complex problems into smaller pieces, which is a top-down structured approach.
- Each function should be a small module, self-contained. It should solve a well defined problem.
- Variable names and function names should be self- explanatory.
- Always comment the code.

vubookshoppk.com

Lecture No. 10

Reading Material

Deitel & Deitel – C++ How to Program — Chapter 3
3.7, 3.11, 3.12, 3.14, 3.17

Contents

- Header Files
- Scope of Identifiers
- Functions
  - Call by Value
  - Call by Reference

vubookshoppk.com

Header Files

You have already been using a header file from day-zero. You know that we used to write at the top before the start of the *main()* function <iostream.h>, with '.h' as an extension, you might have got the idea that it is a header file.

Now we will see why a Header file is used.

In the previous lecture, we discussed a little bit about Function Prototypes. One thing is Declaration and other is Definition. Declaration can also be called as 'Prototype'. Normally, if we have lot of functions and want to use them in some other function or program, then we are left with only one way i.e. to list the prototypes of all of them before the body of the function or program and then use them inside the function or program. But for frequent functions inside a program, this technique increases the complexity (of a program). This problem can be overcome by putting all these function prototypes in one file and writing a simple line of code for including the file in the program. This code line will indicate that this is the file, suppose 'area.h' containing all the prototypes of the used functions and see the prototypes from that file. This is the basic concept of a header file.

So what we can do is:
Make our own header file which is usually a simple text file with '.h' extension ('.h' extension is not mandatory but it is a rule of good programming practice).

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Write function prototypes inside that file. (Recall that prototype is just a simple line of code containing return value, function name and an argument list of data types with semi-colon at the end.)
That file can be included in your own program by using the '#include' directive and that would be similar to explicitly writing that list of function prototypes.

Function prototypes are not the only thing that can be put into a header file. If you remember that we wrote a program for calculating Area of a Circle in our previous lectures. We used the value of 'pi' inside that and we have written the value of 'pi' as 3.1415926. This kind of facts are considered as Universal Constants or Constants within our domain of operations . It would be nice, if we can assign meaningful names to them. There are two benefits of doing this. See, We could have declared a variable of type double inside the program and given a name like 'pi':

```
double  pi  =  3.1415926;
```

Then everywhere in the subsequent calculations we can use 'pi'.
But it is better to pre-define the value of the constant in a header file (one set for all) and simply including that header file, the constant 'pi', is defined. Now, this meaningful name 'pi' can be used in all calculations instead of writing the horrendous number 3.1415926 again and again.

There are some preprocessor directives which we are going to cover later. At the moment, we will discuss about '#define' only. We define the constants using this preprocessor directive as:

```
#define pi  3.1415926
```

The above line does a funny thing as it is not creating a variable. Rather it associates a name with a value which can be used inside the program exactly like a variable. (Why it is not a variable?, because you can't use it on the left hand side of any assignment.). Basically, it is a short hand, what actually happens. You defined the value of the 'pi' with '#define' directive and then started using 'pi' symbol in your program. Now we will see what a compiler does when it is handed over the program after the writing process. Wherever it finds the symbol 'pi', replaces the symbol with the value 3.1415926 and finally compiles the program.
Thus, in compilation process the symbols or constants are replaced with actual values of them. But for us as human beings, it is quite readable to see the symbol 'pi'. Additionally, if we use meaningful names for variables and see a line '2 * pi * radius', it becomes obvious that circumference of a circle is being calculated. Note that in the above statement, '2 * pi * radius'; 2 is used as a number as we did not define any constant for it. We have defined 'pi' and 'radius' but defining 2 would be over killing.

Scope of Identifiers

An 'Identifier' means any name that the user creates in his/her program. These names can be of variables, functions and labels. Here the scope of an identifier means its visibility. We will focus Scope of Variables in our discussion.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Suppose we write the function:

```
void func1()
{
        int i;
        . . .               //Some other lines of code
        int j = i+2;        //Perfectly alright
        . . .
}
```

Now this variable ‘i’ can be used in any statement inside the function func1(). But consider this variable being used in a different function like:

```
void func2()
{
        int k = i + 4;    //Compilation error
        . . .
}
```

The variable ‘i’ belongs to func1() and is not visible outside that. In other words, ‘i’ is local to func1().

To understand the concept of scope further, we have to see what are **Code Blocks**? A code block begins with ‘{‘ and ends with ‘}’.Therefore, the body of a function is essentially a code block. Nonetheless, inside a function there can be another block of code like 'for loop' and 'while loop' can have their own blocks of code respectively. Therefore, there can be a hierarchy of code blocks.

A variable declared inside a code block becomes the local variable for that for that block. It is not visible outside that block. See the code below:

```
void func()
        {
                int outer;                      //Function level scope
                . . .
                {
                        int inner;              //Code block level scope
                        inner = outer;          //No problem
. . .
                }
                inner ++;                       //Compilation error
        }
```

Please note that variable ‘outer’ is declared at function level scope and variable ‘inner’ is declared at block level scope.

The ‘inner’ variable declared inside the inner code block is not visible outside it . In other words, it is at inner code block scope level. If we want to access that variable outside its code block, a compilation error may occur.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

What will happen if we use the same names of variables at both function level scope and inner block level scope? Consider the following code:

```
Line
1.    void increment()
2.    {
3.          int num;                    //Function level scope
4.          . . .
5.          {
6.                int num;              //Bad practice, not recommended
7.                . . .
8.                num ++;               //inner num is incremented
9.                . . .
10.         }
11.   }
```

Note that there is no compilation error if the variable of the same name 'num' is **declared** at line 6 inside the inner code block (at block level scope). Although , there is no error in naming the variables this way, yet this is not recommended as this can create confusion and decrease readability. It is better to use different names for these variables.

Which variable is being **used** at line 8? The answer is the 'num' variable declared for inner code block (at block level scope). Why is so? It is just due to the fact that the outer variable 'num' (at function level scope) is hidden in the inner code block as there is a local variable of the same name. So the local variable 'num' inside the inner code block over-rides the variable 'num' in the outer code block.

Remember, the re-use of a variable is perfectly alright as we saw in the code snippet above while using 'outer' variable inside the inner code block. But re-declaring a variable of the same name like we did for variable 'num' in the inner code block, is a bad practice.

Now, is there a way that we declare a variable only once and then use it inside all functions. We have already done a similar task when we wrote a function prototype outside the body of all the functions. The same thing applies to declaration of variables. You declare variables outside of a function body (so that variable declarations are not part of any function) and they become visible and accessible inside all functions of that file. Notice that we have just used a new word 'file'.
A file or a source code file with extension '.c' or '.cpp' can have many functions inside. A file will contain one *main()* function maximum and rest of the functions as many as required. If you want a variable to be accessible from within all functions, you declare the variable outside the body of any function like the following code snippet has declared such a variable 'size' below.

```
#include <iostream.h>
. . .
// Declare your global variables here
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
int size;
        . . .
        int main( … )
        {
                . . .
        }
```

Now, this ‘size’ is visible in all functions including *main().* We call this as 'file scope variable' or a 'global variable'. There are certain benefits of using global variables. For example, you want to access the variable ‘size’ from anywhere in your program but it does have some pitfalls. You may inadvertently change the value of the variable ‘size’ considering it a local variable of the function and cause your program to behave differently or affect your program logic.

Hence, you should try to minimize the use of global variables and try to use the local variables as far as possible. This philosophy leads us to the concept of Encapsulation and Data Hiding that encourages the declaration and use of data locally.

In essence, we should take care of three levels of scopes associated with identifiers: global scope, function level scope and block level scope.

Let's take a look of very simple example of global scope:

```
        #include <iostream.h>

        //Declare your global variables here
        int i;

        void main()
        {
                i = 10;
cout << “\n” << “In main(), the value of i is: “ << i;
f();
cout << “\n” << “Back in main(), the value of i is: “ << i;
        }

        void f()
        {
                cout << “\n” << ”In f(), the value of i is: “ << i;
                i = 20;
        }
```

Note the keyword ‘void’ here, which is used to indicate that this function does not return anything.

The output of the program is:
In main(), the value of i is: 10
In f(), the value of i is: 10

vubookshoppk.com

Back in main(), the value of i is: 20

Being a global variable, 'i' is accessible to all functions. Function f() has changed its value by assigning a new value i.e. 20.
If the programmer of function f() has changed the value of 'i' accidentally taking it a local variable, your program's logic will be affected.

Function Calling

We have already discussed that the default function calling mechanism of C is a 'Call by Value'. What does that mean? It means that when we call a function and pass some arguments (variables) to it, we are passing a copy of the arguments (variables) instead of original variables. The copy reaches to the function that uses it in whatever way it wants and returns it back to the calling function. The passed copy of the variable is used and original variable is not touched. This can be understood by the following example.

Suppose you have a letter that has some mistakes in it. For rectification, you depute somebody to make a copy of that letter, leave the original with you and make corrections in that copy. You will get the corrected copy of the letter and have the unchanged original one too. You have given the copy of the original letter i.e. the call by value part.
But if you give the original letter to that person to make corrections in it, then that person will come back to you with the changes in the original letter itself instead of its copy. This is call by reference.

The default of C is 'Call by Value'. It is better to use it as it saves us from unwanted side effects. Relatively, 'Call by Reference' is a bit complex but it may be required sometimes when we want the actual variable to be changed by the function being called.

Let's consider another example to comprehend 'Call by Value' and how it works. Suppose we write a *main()* function and another small function *f(int)* to call it from *main()*. This function *f( )* accepts an integer, doubles it and returns it back to the *main()* function. Our program would look like this:

```
#include <iostream.h>

void f(int);                    //Prototype of the function

void main()
{
        int i;
        i = 10;
        cout << "\n" << " In main(), the value of i is: " << i;
        f(i);
        cout << "\n" << " Back in main(), the value of i is: " << i;
}
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
void f (int i)
{
        i *= 2;
        cout << "\n" << " In f(), the value of i is: " << i;
}
```

The output of this program is as under:
In main(), the value of i is: 10
In f(), the value of i is: 20
Back in main(), the value of i is: 10

As the output shows the value of the variable 'i' inside function *main()* did not change, it proves the point that the call was made by value.

If there are some values we want to pass on to the function for further processing, it will be better to make a copy of those values , put it somewhere else and ask the function to take that copy to use for its processing. The original one with us will be secure.

Let's take another example of call by value, which is bit more relevant. Suppose we want to write a function that does the square of a number. In this case, the number can be a double precision number as seen below:

vubookshoppk.com

```
#include  <iostream.h>

double square (double);

void main()
        {
                double num;
                num = 123.456;

                cout << "\n" << " The square of  " << num << " is " << square(num);
                cout << "\n" << " The current value of num is " << num;
        }

double square (double x)
        {
                return x*x;
        }
```

'C' does not have built-in mathematical operators to perform square, square root, log and trigonometric functions. The C language compiler comes along a complete library

for that. All the prototypes of those functions are inside '<math.h>'. In order to use any of the functions declared inside '<math.h>', the following line will be added.

```
#include <math.h>
```

Remember, these functions are not built-in ones but library is supplied with the C-compiler. It may be of interest to you that all the functions inside '<math.h>' are called by value. Whatever variable you will pass in as an argument to these functions, nothing will happen to the original value of the variable. Rather a copy is passed to the function and a result is returned back, based on the calculation on that copy.

Now, we will see why Call by Reference is used.
We would like to use 'call by reference' while using a function to change the value of the original variable. Let's consider the *square(double)* function again, this time we want the original variable 'x' to be squared. For this purpose, we passed a variable to the *square()* function and as a result, on the contrary to the 'Call by Value', it affected the calling functions original variable. So these kinds of functions are 'Call by Reference' functions.

Let us see, what actually happens inside Call by Reference?
As apparent from the name 'By Reference', we are not passing the value itself but some form of reference or address. To understand this, you can think in terms of variables which are names of memory locations. We always access a variable by its name (which in fact is accessing a memory location), a variable name acts as an address of the memory location of the variable.

If we want the called function to change the value of a variable of the calling function, we must pass the address of that variable to the called function. Thus, by passing the address of the variable to the called function, we convey to the function that the number you should change is lying inside this passed memory location, square it and put the result again inside that memory location. When the calling function gets the control back after calling the called function, it gets the changed value back in the same memory location.

In summary, while using the call by reference method, we can't pass the value. We have to pass the memory address of the value. This introduces a new mechanism which is achieved by using '&' (ampersand) operator in C language. This '&' operator is used to get the address of a variable. Let's look at a function, which actually is a modification of our previous *square()* function.

```
#include  <iostream.h>

void square(double*);

        main()
        {
                double x;
                x = 123.456;
                cout << "\n" << " In main(), before calling square(), x = " << x;
                square(&x);                     //Passing address of the variable x
```

```
        cout << "\n" << " In main(), after calling square(), x = " << x;

}

void square(double* x)              //read as: x is a pointer of type double
{
        *x  =  *x  *  *x;           //Notice that there is no space in *x
}
```

Here *x means whatever the x points to and &x means address of the variable x. We will discuss Pointers in detail later.

We are calling function *square(double*)* with the statement *square(&x)* that is actually passing the address of the variable x , not its value. In other words, we have told a box number to the function *square(double*)* and asked it to take the value inside that box, multiply it with itself and put the result back in the same box. This is the mechanism of 'Call by Reference'.

Notice that there is no return statement of *square(double*)* as we are putting the changed value (that could be returned) inside the same memory location that was passed by the calling function.

The output of the program will be as under:
In main(), before calling square(), x = 123.456
In main(), after calling square(), x = 15241.4

By and large, we try to avoid a call by reference. Why? Mainly due to the side-effects, its use may cause. As mentioned above, it will be risky to tell the address of some variables to the called function. Also, see the code above for some special arrangements for call by reference in C language. Only when extremely needed, like the size of the data to be passed as value is huge or original variable is required to be changed, you should go for call by reference, otherwise stick to the call by value convention.

Now in terms of call by reference, we see that there are some places in 'C' where the call by reference function happens automatically. We will discuss this later in detail. For the moment, as a hint, consider array passing in 'C'.

Recursive Function

This is the special type of function which can call itself. What kind of function it would be? There are many problems and specific areas where you can see the repetitive behavior (pattern) or you can find a thing, which can be modeled in such a way that it repeats itself.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Let us take simple example of x^{10}, how will we calculate it? There are many ways of doing it. But from a simple perspective, we can say that by definition $x^{10} = x * x^{9.}$ So what is x^9? It is $x^9 = x * x^8$ and so on.

We can see the pattern in it:
$x^n = x * x^{n-1}$

To compute it, we can always write a program to take the power of some number. How to do it? The power function itself is making recursive call to itself. As a recursive function writer, you should know where to stop the recursive call (base case). Like in this case, you can stop when the power of x i.e. n is 1 or 0.

Similarly, you can see lot of similar problems like Factorials. A factorial of a positive integer 'n' is defined as:

n! = (n) * (n-1) * (n-2) * ….. * 2 * 1

Note that
n! = (n) * (n-1)!
and (n-1)! = (n-1) * (n-2)!

This is a clearly a recursive behavior. While writing a factorial function, we can stop recursive calling when n is 2 or 1.

vubookshoppk.com

```
long fact(long n)
	{
		if (n <= 1)
			return 1;
		else
			return  n * fact(n-1);
	}
```

Note that there are two parts (branches) of the function: one is the base case (which indicates when the function will terminate) and other is recursively calling part.

All the problems can be solved using the iterative functions and constructs we have studied until now. So the question is: do we need to use recursive functions? Yes, it adds little elegance to the code of the function but there is a huge price to pay for this. Its use may lead to the problems of having memory overhead. There may also be stacking overhead as lots of function calls are made. A lot of functions can be written without recursion (iteratively) and more efficiently.

So as a programmer, you have an option to go for elegant code or efficient code, sometimes there is a trade-off. As a general rule, when you have to make a choice out of elegance and efficiency, where the price or resources is not an issue, go for elegance but if the price is high enough then go for efficiency.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

'C' language facilitates us for recursive functions like lot of other languages but not all computer languages support recursive functions. Also, all the problems can not be solved by recursion but only those, which can be separated out for base case, not iterative ones.

Tips

Header file is a nice mechanism to put function prototypes and define constants (global constants) in a single file. That file can be included simply with a single line of code.
There are three levels of scopes to be taken care of, associated with identifiers: global scope, function level scope and block level scope.
For Function calling mechanism, go for 'Call by Value' unless there is a need of 'Call by Reference'.
Apply the recursive function where there is a repetitive pattern, elegance is required and there is no resource problem.

vubookshoppk.com

Lecture No. 11

Reading Material

Deitel & Deitel - C++ How to Program	chapter 4 4.2, 4.3, 4.4

Summary

- Introduction
- Arrays
- Initialization of Arrays
- Sample Program 1
- Copying Arrays
- Linear Search
- The Keyword ‘const’
- Tips

vubookshoppk.com

Introduction

We have started writing functions, which will become a part of our every program. As C language is a function-oriented language, so we will be dealing with too many functions. Our programming toolkit is almost complete but still a very important component is missing. We are going to discuss this component i.e. **Arrays** in this lecture.

Let us consider an example about calculation of average age of 10 students. At first, we will declare 10 variables to store the age of each student and then sum up all the ages and divide this with 10 to get the average age. Suppose, we have 100 students instead of 10, we have to declare 100 variables i.e. one for each student's age. Is there any other way to deal with this problem? Arrays are possible solution to the problem.

Array is a special data-type. If we have a collection of data of same type as in the case of storage of ages of 100 students, arrays can be used. Arrays are data structure in which identical data types are stored. The concept of arrays is being explained further in the following parts of the lecture.

Arrays

In C language, every array has a data type i.e. name and size. Data type can be any valid data type. The rules of variable naming convention apply to array names. The size of the array tells how many elements are there in the array. The size of the array should be a precise number. The arrays occupy the memory depending upon their size and have contiguous area of memory. We can access the arrays using the array index.

Declaration:

The declaration of arrays is as follows:

```
data_type       array_name [size] ;
```

for example:

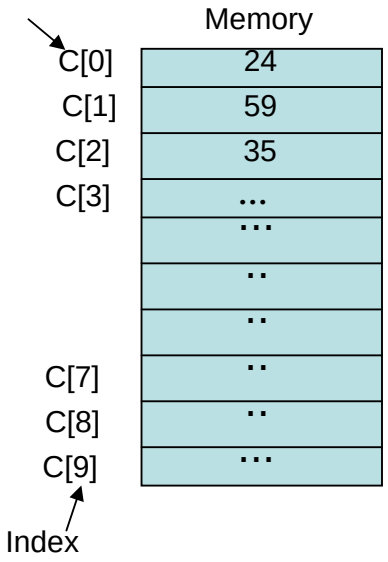
```
int ages[10];
```

Let's consider an array int C[10]; This is an array of integer and has a name 'C'. It has a size ten which depicts that the array 'C' can contain ten elements of int data type. In the memory, the array occupies the contiguous area, in this case it will occupy forty bytes (one int = 4 bytes). The elements of the array are manipulated using the index. In C language, the index of array starts from zero and is one less than array's size. Index of array is also called subscript.

Memory image of an array:

Name

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533



In the above figure, the memory chunk containing the array C is shown. On the first line, C[0] is written while on the 2nd line, C[1] is written and so on. The number in the [ ] is the index of the array. C[0] is used for the first element, followed by C[1] for the second element and so on. It is important to note that in an array the index 6 ([6]) means the seventh element of the array and thus the eighth element will have an index 7. Thus, the index of the last element of the array will be 1 less than the size of the array. On the right hand side, the values of the elements are shown in the memory i.e. the value of the element at zero position (C[0]) is 24 while that of the element at first position (C[1]) is 59 and so on. The important thing to be noted here is that the indexing of the array starts from zero, not from one. So in the above example, the index of the array C will be from C[0] to C[9]. If we have an array of size 25, its index will be from 0 to 24.

Usage of Arrays

To declare arrays, we have to give their data type, name and size. These are fixed-size arrays. In the coming lectures, we will discuss arrays without using size at declaration time. Arrays may be declared with simple variables in a single line.

```
int  i, age [10];
int height [10], length [10] ;
```

To access array, we can't use the whole array at a time. We access arrays element by element. An index (subscript) may be used to access the first element of the array. In this case, to access first element we write like age[0]. To access the 5th element, we will write age[4] and so on. Using the index mechanism, we can use the array elements as simple variables. Their use can be anywhere where there we can use a simple variable i.e. in assignment statements, expressions etc. Please do not confuse the usage of array and declaration of array. When we write int age [10], it means we are declaring an array of type int, its name is age and its size is 10. When we write

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

age[5], it means we are referring to the single element of the array not the whole array.

Consider the example of student's ages again. Is there a way to calculate the average age of all the students in an array?
As we know that arrays can be accessed with indexing. So we can use a '*for* loop' as under;

```
for (i = 0 ; i < 10 ;  i++ )
{
        cout << "Please enter the age of the student ";
        cin >> age [i];
}
```

In the above '*for loop*' the value of *i* is changing from 0 to 9. Here the loop condition is *i<10.* This means that the *cin* and *cout* statements will be executed 10 times. We have used *i* as the index of the array. The index we are referring to the array needs to be an integer. It can be 4, 5 or an integer variable like *i*. In the first repetition, the value of *i* is 0, i.e. *age[0]* so the value of first element of the age will be read. In the second repetition, the value of *i* becomes 1 i.e. *age[1]* so the value of 2nd element of the age will be read and so on. We get all the 10 values from the user which will be stored in the array *age*.

vubookshoppk.com

Now we will calculate the total of ages. We can use another '*for loop*' to add up all the elements of the array age.

```
int totalAge = 0;

for (i = 0 ; i < 10 ;  i++ )
{
        totalAge += age [i];
}
```

In the above loop, all the elements of the array age will be added to the variable *totalAge*. When the value of *i* is 0 i.e. *age[0]* the value of first element will be added to the *totalAge*. As the value of *i* is changing from 0 to 9 so all the 10 elements of the array will be added to the *totalAge*. By dividing this *totalAge* by 10 we will get the average age.

Initialization of Arrays

There are many ways to initialize an array. Don't use the default initialization of arrays. Compiler may assign some value to each declared array. Always initialize the array in such a manner that the process is clear.

We can initialize an array using a 'loop' while assigning some value.

```
int i, age [10];
```

```
for ( i = 0; i < 10 ; i++ )
{
        age[i] = 0;
}
```

With the help of this simple loop, we have initialized all the elements of array *age* to zero. In the loop condition, we have used the condition *i < 10,* where the size of the array is ten. As we know, the array index is one less than the size of the array. Here we are using *i* as the index of array and its values are from 0 to 9.

We can also initialize the array at the time of declaration as:

```
int age [10] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
```

The above statement creates an array *age* of integers and initializes all the elements with zero. We can use any value to initialize the array by using any other number instead of zero. However, generally, zero is used to initialize the integer variables. We can do it by using the following shortcut.

```
int age [10] = { 0 };
```

vubookshoppk.com

The above statement has also initialized all the elements of the array to zero.

We have different ways of initializing the arrays. Initialization through the use of loop is a better choice. If the size of the array gets larger, it is tedious to initialize at the declaration time.

Consider the following statement:

```
int age [ ] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
```

Here we have not mentioned the size of the array. The compiler is quite intelligent as it detects the initialization list which consists of ten 0's. Therefore, it creates an array of 10 integers and initializes all the elements with zero.

The index of the arrays starts from the index 0 and is up to one less than the size of the array. So if the size of the array is ten, the index will be from 0 to 9. Similarly, if the size of the array is 253, the index will be from 0 to 252.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Sample Program 1

Problem Statement:

Write a program which reads integers from the user and stores these ones in an array. User can enter a maximum of 100 numbers. Stop taking input when user enters -1.

Solution:

We have to declare an integer array of size 100 to be used to store the integers. We used a loop to get the input from the users. There are two conditions to terminate the loop i.e. either user has entered 100 numbers or user entered -1. 'For' and 'while' loops can execute zero or more times whereas 'do-while' may execute one or more times. By analyzing the problem, the loop will be executed at least once so do-while loop logically fits in this problem. We take an integer *z* to get the input from the user and *i* as the counter so the condition will be as (*z != -1 && i < 100*). *&&* is used to enforce that both the conditions are true. If any of the two conditions becomes false, the loop will be terminated. The loop counter is less than 100 because the index of the array will be from 0 to 99.

We will read a number from the user and store it at some particular location of the array unless user enters -1 or 100 numbers are entered. In the loop, we will use the *if statement* whether the number entered by user is -1 or not. If the number entered is not -1, then we will store it in the array. The index of the array will also be incremented in each repetition. We can assign some value to array element as:

```
c[ 3 ] = 33;
```

In an assignment statement, we cannot use expression on the left hand side. Here c[3] is used as a variable which represents the 4th element of the array.

The complete code of the program as under:

```
// This program reads the input from user and store it into an array and stop at -1.

#include  <iostream.h>

main( )
{
        int  c [ 100 ] ;
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
        int z , i = 0 ;
        do
        {

                cout  << "Please enter the number (-1 to end input) " << endl;
                cin >> z ;
                if ( z != -1 )
                {
                        c[ i ] = z ;
                }
                i ++ ;
        } while ( z != -1 && i < 100 ) ;

        cout << " The total number of positive integers entered by user is " << i -1;
}
```

The above code shows that the assignment statement of the array is inside the *if* block. Here the numbers will be assigned to the array elements when the 'if statement' evaluates to true. When the user enters -1, the *if statement will* evaluate it false. So the assignment statement will not be executed and next *i* will be incremented. The condition in the 'while loop' will be tested. As the value of *z* is -1, the loop will be terminated.
Now we have to calculate how many positive numbers, the user has entered. In the end, we have incremented *i* so the actual positive integers entered by the users is *i -1.*
The above example is very useful in terms of its practical usage. Suppose we have to calculate the ages of students of the class. If we don't know the exact number of students in the class, we can declare an array of integers of larger size and get the ages from the user and use -1 to end the input from the user.

A sample out put of the program is as follow.

```
Please enter the number (-1 to end input)  1
2
3
4
5
6
-1
The total number of positive integers entered by user is  6
```

Copying Arrays

Sometimes, we need to copy an array. That means after copying, both the arrays will contain elements with same values. For being copy able, both arrays need to be of same data type and same size. Suppose, we have two arrays **a** and **b** and want to copy array **a** into array **b**. Both arrays are of type **int** and of size 10.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
int array a[10];
int array b[10];
```

We know that a value can be assigned to an element of array using the index. So we can write assignment statements to copy these arrays as:

```
b[0] = a[0] ;
b[1] = a[1] ;
b[2] = a[2] ;
……
……
……
b[9] = a[9] ;
```

As the size of array is 10, its index will be from 0 to 9. Using the above technique, we can copy one array to another. Now if the array size is 100 or 1000, this method can be used. Is there some other way to do things in a better way? We can use the loop construct to deal with this easily in the following way.

```
for (i = 0; i < 10 ; i ++)
{
        b[i] = a[i];
}
```

With the help of loop, it becomes very simple. We are no more worried about the size of the array. The same loop will work by just changing the condition. We are assigning the corresponding values of array **a** into array **b**. The value of first element of array **a** is assigned to the first element of array **b** and so on.

Example:

Take the sum of squares of 10 different numbers stored in an array.

Here is the code of the program:

```
// This program calculates the sum of squares of numbers stored in an array.

#include <iostream.h>

main()
{
        int a[10];
        int sumOfSquares = 0 ;
        int i =0;

        cout << "Please enter the ten numbers one by one " << endl;

        // Getting the input from the user.
        for (i = 0 ; i < 10 ;  i++ )
        {
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
                cin >> a [i];
        }

        // Calculating the sum of squares.
        for ( i = 0 ; i < 10 ; i ++ )
        {
                sumOfSquares = sumOfSquares + a[ i ] * a[ i ] ;
        }

        cout << "The sum of squares is  " << sumOfSquares << endl;
}
```

A sample out put of the program is given below.

```
Please enter the ten numbers one by one
1
2
3
4
5
6
7
8
9
10
The sum of squares is 385
```

vubookshoppk.com

Linear Search

Arrays are used to solve many problems. As we have seen that loops are used along with the arrays, so these two constructs are very important. Suppose, we are given a list of numbers to find out a specific number out of them. Is the number in the list or not? Let's suppose that there are 100 numbers in the list. We take an array of size 100 as *int a [100]. For populating it,* , we can request the user to enter the numbers. Either these numbers can be stored into the array or we can just populate it with numbers from 0 to 99. We can write a simple loop and assign the values as *a[i] = i.* This means that at ith position, the value is *i* i.e. (a[5] = 5), at 5th position the value is 5 and so on. Then we can request the user to enter any number and store this number into an int variable. To search this number in the array, we write a loop and compare all the elements with the number. The loop will be terminated, if we found the number or we have compared all the elements of the array, which means that number is not found. We used a flag to show that we have found the number or not. If the value of found is zero, the number is not found while the value 1 will mean that number has been found. When we find the number, is there a need to compare it with other elements of the array? May be not, so when we found the number, we just jumped out of the loop. In the end, we check the variable *found*. If the value is 1, it means number has been found. Otherwise number stands unfound.

Here is the complete code of the program.

```
// This program is used to find a number from the array.
#include <iostream.h>

main()
{
        int z, i ;
        int a [ 100 ] ;
        // Initializing the array.
        for ( i =0 ; i < 100 ; i ++ )
        {
                a [ i ] = i ;
        }

        cout << " Please enter a positive integer   " ;
        cin >> z ;
        int found = 0 ;

        // loop to search the number.
        for ( i = 0 ; i < 100 ; i ++ )
        {
                if ( z == a [ i ] )
                {
                        found = 1 ;
                        break ;
                }
        }
        if ( found == 1 )
                cout << " We found the integer at index " << i ;
        else
                cout << " The number was not found " ;
}
```

vubookshoppk.com

The following is an output of the program.

```
Please enter a positive integer   34
We found the integer at index  34
```

The loop in the above program may run 100 times or less. The loop will terminate if the number is found before the 100th repetition. Therefore, in the linear search the maximum limit of the loop execution is the size of the list. If the size of list is 100, then the loop can execute a maximum of 100 times.

Using random function (Guessing Game):

We can turn this problem into an interesting game. If we as programmers do not know, which number is stored in the array? We can make this a guessing game. How can we do that? We need some mechanism by which the computer generates some number. In all the C compilers, a random number generation function is provided. The function is *rand()* and is in the standard library. To access this function, we need to

include <stdlib.h> library in our program. This function will return a random number. The number can be between 0 and 32767. We can use this function as:

x = rand ( );

The random function generates an integer which is assigned to variable x. Let's consider the function-calling mechanism. The program starts its execution in the main function. When the control goes to the statement containing a function call, the main program stops here and the control goes inside the function called. When the function completes or returns some value, the control comes back to the main program.

Here is the complete code of the program using rand().

```
// This program is used to find a number from the array.

#include <iostream.h>
#include <stdlib.h>

main()
{
        int z, i ;
        int a [ 100 ] ;
        // Initializing the array.

        for ( i =0 ; i < 100 ; i ++ )
        {
                a [i] = rand() ;
        }

        cout << " Please enter a positive integer " ;
        cin >> z ;
        int found = 0 ;

        // loop to search the number.
        for ( i = 0 ; i < 100 ; i ++ )
        {
                if ( z == a [ i ] )
                {
                        found = 1 ;
                        break ;
                }
        }
        if ( found == 1 )
                cout << " We found the integer at position " << i ;
        else
                cout << " The number was not found " ;
}
```

vubookshoppk.com

The following is an output of the program.

```
Please enter a positive integer   34
```

The number was not found

The function rand ( ) returns a value between 0 and 32767. Can we limit the generated random number in a smaller range? Suppose we have a die with six faces marked with 1, 2, 3, 4, 5 and 6. We want to generate random die number i.e. the number should be between 1 and 6 inclusive. Here we can use the modulus operator to achieve this. Modulus operator returns the remainder. What will be the result of the statement?

```
rand ( ) % 6
```

When 6 divides any number, the remainder will always be less than 6. Therefore, the result will be between 0 and 5 inclusive. We want the number between 1 and 6, therefore we will add 1.

```
1 + rand ( ) % 6;
```

The above statement will give us the desired result. We need to know whether this is a fair die or not. A fair die is a die when it is rolled 10 or 100 million of times. Then on average, equal number of 1's, equal number of 2's, equal number of 3's etc. will be generated. Can we test our die i.e. it is fair or not? That is there are equal numbers of chances of 1 or 2 etc. Think about generating a test for our random number generator. Does it produce a fair die?

The random function is very useful. It can be used to guess the tossing of the coin. There can be only two possibilities of tossing a coin. Therefore we can use rand ( ) % 2 which will give 0 or 1.

The Keyword 'const':

To declare an array, we need its data type, name and size. We use simple integer for the size like 10 or 100. While using arrays in loops, we use the size a lot. Suppose if we have to change the size of the array from 10 to 100, it will have to be changed at all the places. Missing a place will lead to unexpected results. There is another way to deal this situation i.e. keyword construct. The keyword *const* can be used with any data type and is written before the data type as:

```
const int arraySize = 100;
```

This statement creates an identifier *arraySize* and assigns it the value 100. Now the *arraySize* is called integer constant. It is not a variable. We cannot change its value in the program. In the array declaration, we can use this as:

```
int age [arraySize];
```

Now in the loop condition, we can write like this:

```
for ( i = 0; i < arraySize ; i ++)
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

If we have to change the size of the array, we only have to change the value of *arraySize* where it is declared. The program will work fine in this case. This is a good programming practice to use *const* for array size.

Tips

Initialize the array explicitly
Array index (subscript) starts from 0 and ends one less than the array size
To copy an array, the size and data type of both arrays should be same
Array subscript may be an integer or an integer expression
Assigning another value to a const is a syntax error

vubookshoppk.com

Lecture No. 12

Reading Material

Deitel & Deitel - C++ How to Program chapter 4
4.4, 4.5, 4.6, 4.8, 4.9

Summary

- Character Arrays
- Initialization Of Character Arrays
- Arrays Comparison
- Sorting Arrays
- Searching arrays
- Functions And arrays
- Example 1
- Multidimensional Arrays
- Example 2
- Tips

vubookshoppk.com

Character Arrays

While dealing with words and sentences, we actually make use of character arrays. Up to now, we were dealing with integer arrays and storing integer values. Here we have to see what needs to be done for storing a name. A simple variable can't be used to store a name (which is a string of characters) as a variable stores only a single character. We need a character array to grab a name. A character array is not different from an integer array. To declare a character array, we will write as under:

char name [100] ;

In this way, we declare a string or character array. There are some special properties of character arrays. Suppose that we declare an array of 100 characters. We enter a name with 15-20 characters. These characters in the array occupy 15-20 character spaces. Now we have to see what has happened to the remaining character spaces in the array. Similarly, a question arises, will an array displayed on the screen, show 100 characters with a name in 15-20 spaces and blanks for the remaining. Here C has a character handling capability i.e. the notion of strings. When we place a string in a character array, the computer keeps a mark to identify that the array was of this size while the string stored in it is of the other size. That marker is a special character, called **null** character. The ASCII code of **null** character is all zeros. In C language, we represent the **null** character as "**\0**". C uses this character to terminate a string. All strings are terminated with the **null** character.

Now, we will see how the character arrays are stored in memory. While declaring a character array, we normally declare its size larger than the required one. By using a character array, it becomes easy to store a string. We declare a character array as under.

char name [100] ;

Now we can store a string in this array simply by using the *cin* statement in the following way.

cin >> name ;

In the above statement, there is an array on right hand side of *cin* instead of a simple variable. The *cin* stream has a built-in intelligence that allows the compiler (program) to read whole string at a time rather than a single character as in case of simple variable of type *char*. The compiler determines that the *name* is not a simple variable. Rather it is a string or character array. Thus *cin* reads a character array until the user presses the enter key. When enter key is pressed, *cin* takes the whole input (i.e. string) and stores it into the array *name*. The C language, by itself, attaches a *null* character at the end of the string. In this way, the total number of spaces occupied in the array by the string is the number of characters entered by the user plus 1 (this one character is the **null** character inserted at the end of the string by C automatically). The **null** character is used to determine where the populated area of the array has ended. If we put a string larger than the size of the array in absence of a *null* character in it, then it is not possible to determine where a string is terminated in the memory. This can cause severe logical error. So, one should be careful while declaring a character array. The size of array should be one more than the number of characters you want to store.

Initialization Of Character Arrays

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Now we will look into integer array initialization process that can provide a list of integer values separated by commas and enclosed in curly braces. Following is the statement through which we initialize an integer array.

```
int age [5] = {12, 13, 16, 13, 14};
```

If we don't mention the size of the array and assign a list of values to the array, the compiler itself generates an array of the size according the number of values in the list. Thus, the statement int age [] = {14, 15, 13}; will allocate a memory to the array of size 3 integers. These things also apply to character arrays as well. We can initialize an array by giving a list of characters of the string, the way we assign integer values in integer array. We write the characters of this string one by one in single quotes (as we write a single character in single quotes), separated by commas and enclosed in curly braces. So the initialization line will be as under

```
char name [100] = {'i', 'm', 'r', 'a', 'n'};
```

we can also write the string on right hand side in double quotes as

```
char name [100] = "imran" ;
```

The easy way to initialize a character array is to assign it a string in double quotes. We can skip the size of the array in the square brackets. We know that the compiler allocates the memory at the declaration time, which is used during the execution of the program. In this case, the compiler will allocate the memory to the array of size equal to the number of characters in the provided string plus 1 (1 is for the *null* character that is inserted at the end of string). Thus it is a better to initialize an array in the following way.

```
char name [] = "Hello World" ;
```

In the above statement, a memory of 12 characters will be allocated to the array *name* as there are 11 characters in double quotes (space character after Hello is also considered and counted) while the twelfth is the null character inserted automatically at the end of the string.

We can do many interesting things with arrays. Let's start with reading a string (for example your name) from keyboard and displaying it on the screen. For this purpose, we can write the following code segment

```
char name [100] ;
cout << "Please enter your name :  " ;
cin >> name ;
```

In the *cin* statement, when the user presses the enter key the previous characters entered, that is a string will be stored in the array *name*. Now we have a string in the array *name*. We can display it with *cout* statement. To display the string, we have stored in *name.* We can write as under

```
cout << name ;
```

This will display the string. Alternatively, we can use a loop to display the string. As the string is an array of characters, we can display these characters one by one in a '*for* loop'. We can write a loop as under

```
for ( i = 0 ; i < 100 ; i ++ )
cout << name [ i ] ;
```

Thus this loop will display the characters in the array one by one in each iteration. First, it will display the character at name [0], followed by that at name [1] and so on. Here we know that the string in the array is terminated by a **null** character and after this **null** character, there are random values that may not be characters (some garbage data) in the array. We don't want to display the garbage data that is in the array after this **null** character. While using the statement *cout << name;* the *cout* stream takes

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

the characters of the array *name* up to the **null** character and the remaining part of the array is ignored. When we are displaying the characters one by one, it is necessary to stop the displaying process at the end of a string (which means when null character is reached). For this purpose, we may put a condition in the loop to terminate the loop when the null character is reached. So we can use *if* statement in the loop to check the **null** character. We can modify the above *for* loop so that it could terminate when **null** character reaches in the array.

```
for ( i = 0 ; i < 100 ; i ++ )
{     if (name [ i ] == '\0')
            break ;

cout << name [ i ] ;
}
```

Here a while loop can also be used instead of a 'for loop'.

Arrays Comparison

We can use this character-by-character manipulation of the array to compare the characters of two arrays of the same size. Two arrays can be equal only when first of all their sizes are equal. Afterwards, we compare the values of the two arrays with one to one correspondence. If all the values in the first array are equal to the corresponding values of the second array, then both the arrays will be equal. Suppose, we have two integer arrays **num1** and **num2** of size 100 each and want to find whether both arrays are equal. For this purpose, we will declare a flag and set it to zero, that means that arrays are not equal this time. For this flag, we write *int equals = 0* ;
To compare the values of the arrays one by one, we write a for loop i.e. *for (i = 0 ; i < 100 ; i ++). In* the body of the *for* loop, we use an *if* statement to check the values. In the *if* statement, we use the not equal operator (!=). The advantage of using not-equal operator is that in case if the values at some position are not equal to each other, then we need not to compare the remaining values. We terminate the loop here and say that the arrays are not equal. If the values at a position are equal, we continue to compare the next values. If all the values are found same, we set the flag *equal* to 1 and display the results that both the arrays are identical. The same criterion applies to character arrays. The comparison of character arrays is very common. While finding a name in a database, we will compare two character arrays (strings). The comparison of two strings is so common in programming that C has a function in its library to manipulate it. We will discuss it later in the lecture on string handling. For the time being, we will write our own function to find the equality of two strings.

Following is the code of a program, which takes two arrays of 5 numbers from the user and compares them for equality.

```
// This program takes two arrays of 5 integers from user
//displays them and after comparing them displays the result
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
# include <iostream.h>

main ( )
{
        int num1 [5], num2 [5], i, equals = 0 ;
        // input of 5 integers of first array
        cout << "Please enter five integers for the first array" << endl ;
for ( i = 0 ; i < 5 ; i ++)
                cin >> num1 [ i ] ;

        // input of 5 integers of 2nd array
        cout << "Please enter five integers for the second array" << endl ;
for ( i = 0 ; i < 5 ; i ++)
                cin >> num2 [ i ] ;

        //display the elements of two arrays
        cout << "\n The values in the first array are  :   " ;
for ( i = 0 ; i < 5 ; i ++)
                cout << "\t" << num1 [ i ] ;

        cout << "\n The values in the second array are :   " ;
for ( i = 0 ; i < 5 ; i ++)
                cout << "\t" << num2 [ i ];

        // compare the two arrays
        for ( i = 0 ; i < 5 ; i ++ )
        {       if ( num1 [ i ]  != num2 [ i ] )
                {
                        cout << "\n The arrays are not equal " ;
                        equals = 0 ;     //set the flag to false
break ;
                }
                equals = 1;      //set flag to true
        }

        if (equals)
                cout << "\n Both arrays are equal" ;
}
```

vubookshoppk.com

Similarly, we can write a program that compares two strings (character arrays) of the same size. While comparing strings, a point to remember is that C language is case-sensitive. In C-language 'A' is not equal to 'a'. Similarly, the string "AZMAT" is not equal to the string "azmat" or "Azmat".

A sample out-put of the program is given below.

```
Please enter five integers for the first array
1
3
```

```
5
7
9
Please enter five integers for the second array
1
3
4
5
6
The values in the first array are :   1        3       5       7       9
The values in the first array are :   1        3       4       5       6
The arrays are not equal
```

Sorting

We want to sort an array in ascending order. There may be many ways to sort an array. Suppose we have an array of 100 numbers. To sort it in ascending order, we start from the first number (number at zero index) and find the smallest number in the array. Suppose, we find it at sixteenth position (index 15). If we assign this number directly to the first position, the number already placed at first position will be over written. But we want that number should exist in the array. For this purpose, we use a technique called *swapping*. In this technique, we swap two values with each other. For this purpose, we declare a variable and assign the value of first variable to this variable before assigning the second number (i.e. to be swapped) to the first variable. Then we assign the value of second variable to the first variable. Afterwards, the number, which we have stored in a separate third variable (that is actually the value of first variable) is assigned to the second variable. In arrays, the single element of an array is treated as a single variable so we can swap two numbers of an array with each other with this technique.
In our sorting process, we declare a variable *x* and assign it the number at the first position. Then assign the number at sixteenth position to the first position. After this, we assign the number in *x* (that is actually the number that was at first position in the array) to the sixteenth position. In programming, this can be done in the following fashion.

```
x = num [0] ;                    // assign number at first position to x
num [0] = num [15] ;  // assign number at sixteenth position to first position
num [15] = x ;            // assign number in x to sixteenth position
```

We have the smallest number at the first position. Now we start reading the array from second position (index 1) and find the smallest number. We swap this number with the second position before starting from index 2. The same process can be repeated later. We continue this process of finding smallest number and swapping it till we reach the last number of the array. The sorting of array in this way is a brute force and a very tedious work. The computer will do fine with small arrays. The large arrays may slow it down.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Searching

The same applies to the search algorithms. For finding out a particular number in an array, we can use technique of linear search. In this technique, there may be as many comparisons as numbers in the array. We make comparison of the number to be found with each number in the array and find it out if it matches any number in the array. However, we can perform even better by using a binary search algorithm.

Binary Search Algorithm

In binary search algorithm, the 'divide and conquer' strategy is applied. This algorithm applies only to sorted arrays in ascending or descending order. Suppose that we want to search a number in an ascending array. For this purpose, we divide the array into two parts (say left and right). We compare the target value with the value at middle location of the array. If it does not match, we see whether it is greater or less than the middle value. If it is greater than the middle value, we discard the left part of the array. Being an ascending array, the left part contains the smaller numbers than the middle. Our target number is greater than the middle number. Therefore, it will be in the right part of the array. Now we have a sub-array, which is the half of the actual array (right side portion of main array). Now we divide this array into two parts and check the target value. If target value is not found, we discard a portion of the array according to the result whether target value is greater or less than the middle value. In each iteration of testing the target value, we get an array that is half of the previous array. Thus, we find the target value.

The binary search is more efficient than the linear search. In binary search, each iteration reduces the search by a factor of two (as we reduce to half array in each iteration). For example, if we have an array of 1000 elements, the linear search could require 1000 iterations. The binary search would not require more than 10. If an array has elements 2^n, then the maximum number of iterations required by binary search will be n. If there are 1000 elements (i.e. 2^{10}, actually it will 1024), the number of iterations would not be more than 10.

Functions and Arrays

In C language, the default mechanism of calling a function is 'call by value'. When we call a function, say *fn*, and pass it a parameter *x* (argument value) by writing statement fn(x), the calling mechanism puts the value of *x* at some other place. Then calls the function and gives this value to it. This means a copy of the value is sent to the program. The original *x* remains untouched and unchanged at its place. The function uses the passed value (that has placed at some other place) and manipulates it in its own way. When the control goes back to the calling program, the value of original *x* is found intact. This is the call by value mechanism.
Now let's see what happens when we pass an array to a function. To pass an array to a function, we will tell the function two things about the array i.e. the name of the array and the size. The size of the array is necessary to pass to the function. As the array is declared in the calling function, it is visible there. The calling function knows its size but the function being called does not know the size of the array. So it is necessary to pass the size of the array along with its name. Suppose we have declared a character array in the program by the following statement:

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

char name[50] ;

We have a function (say *reverse,* you should write it as an exercise) that reverses the array elements and displays them.

Firstly, we need to write the prototype of the function reverse. We say that this function returns nothing so we use the keyword *void* in its return type. Secondly, we have to write the parameters this function will get. We write these parameters with their type.

Now the prototype of this function will be written as

void reverse (char [], int) ;

In the above statement, the brackets [] are necessary. These brackets indicate that an array of type char will be passed to the function. If we skip these brackets and simply write *char,* it will mean that a single character will be passed to the function. In addition, the second parameter i.e. of type *int*, is of array's size. Note that in the prototype of the function we have not written the names of the parameters. It is not necessary to write the names of the parameters in function prototype. However, if we write the names, it is not an error. The compiler will simply ignore these names.

Now we will define the function reverse. In the function's definition, we will use the array and variable names. These names are local to this function so we can give these variables a name other than the one used in declaration in the calling program. We write this as below.

```
void reverse ( char characters [], int arraySize )
{
    // The body of the function.
}
```

Here, the body of the function is left over for an exercise.

Let's say we have a character array *name* and a name 'adnan' is stored in it. We call the reverse function by passing the array *name* to it. For this we write *reverse (name, 100)*;

In this function call, we are sending the name of the array to the function i.e. *name* and the size of the array that is 100. When this call of the function is executed the control goes to the function *reverse*. The statements in this function are executed which reverses the array and displays it. After this, the control comes back to the main function to the statement next to the function call statement. The return type of the function is *void* so it does not return any thing. Now in the main, we write the statement *cout << name*; What will be displayed by this statement? Whether it will be the original name 'adnan' or something else. It will display the reversed array. In this instance, we see that whatever the function *reverse* did to the array (that was passed to it) is appearing in the calling function. It means that the original array in the calling program has been changed. Here we change (reverse) the order of the characters of array in the function and find that the characters of the array in the calling function are reversed. This means that the called function has not a copy of the array but has the original array itself. Whereas in case of simple variables, a called function uses a copy of variables passed to it in a *'call by value'* mechanism, which is by default in case of simple variables. In arrays, the by default mechanism is '*call by reference'*. While passing arrays to a function, we don't need to use & and * operators, as we use for variables in *call by reference* mechanism.

Thus if we pass an array to a function, the array itself is passed to the function. This is due to the fact that when we declare an array, the name of the array has the address of

vubookshoppk.com

the memory location from where the array starts. In other words, it is the address of the first element of the array. Thus the name of the array actually represents the address of the first location of the array. Passing the name of array to a function means the passing of the address of the array which is exactly the same as we do in *call by reference*. So whatever the function does to the array, it is happening in the same memory locations where the array originally resides. In this way, any modifications that the function does to the contents of the array are taking place in the contents of the original array too. This means that any change to the array made by the function will be reflected in the calling program. Thus an important point to remember is that whenever we pass simple variables to a function, the default mechanism is *call by value* and whenever we pass an array to a function, the default mechanism is *call by reference*. We know that when we talk about a single element of an array like x [3] (which means the fourth element of the array x), it is treated as simple variable. So if we pass a single element of an array to a function (let's say like fn (x [3]);), it is just like a simple variable whose copy is passed to the function (as it is a call by value). The original value of the element in the array remains the same. So be careful while passing arrays and a single element of array to functions. This can be well understood from the following examples.

Example 1

Suppose we declare an array in the main program and pass this array to a function, which populates it with values. After the function call, we display the elements of the array and see that it contains the values that were given in the function call. This demonstrates that the called function changes the original array passed to it.

Following is the code of the program.

```
//This program demonstrates that when an array is passed to a function then it is a call by
//reference and the changes made by the function effects the original array

# include <iostream.h>

void getvalues( int [], int) ;

main ( )
{
        int num [10], i ;
        getvalues ( num, 10) ; //function call, passing array num
        //display the values of the array
        cout << "\n The array is populated with values \n" ;
        for ( i = 0 ; i < 10 ; i ++)
cout << " num[" << i << "] = " << num[i]<< endl ;
}
void getvalues ( int num[], int arraysize)
{
        int i ;
for ( i = 0 ; i < arraysize ; i ++)
        num[i] = i ;
 }
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Here in the function *getvalues*, we can get the values of the array from user by using the *cin* statement.

Following is the output of the execution of the program.

```
The array is populated with values
num[0] = 0
num[1] = 1
num[2] = 2
num[3] = 3
num[4] = 4
num[5] = 5
num[6] = 6
num[7] = 7
num[8] = 8
num[9] = 9
```

Multidimensional Arrays

There may be many applications of arrays in daily life. In mathematics, there are many applications of arrays. Let's talk about vectors. A vector is a set of values which have independent coordinates. There may be two-dimensional vector or three-dimensional vector. There are dot and cross products of vectors besides many other manipulations. We do all the manipulations using arrays. We manipulate the arrays with loops. Then there is a mathematical structure matrix, which is in rows and columns. These rows and columns are manipulated in two-dimensional arrays. To work with rows and columns, C provides a structure i.e. a two-dimensional array. A two dimensional array can be declared by putting two sets of brackets [] with the name of array. The first bracket represents the number of rows while the second one depicts the number of columns. So we can declare an array *numbers* of two rows and three columns as follows.

```
int numbers [2] [3] ;
```

Using two-dimensional arrays, we can do the addition, multiplication and other manipulations of matrices. A value in a two-dimensional array is accessed by using the row number and column number. To put values in a two-dimensional array is different from the one-dimensional array. In one-dimensional array, we use a single '*for* loop' to populate the array while nested loops are used to populate the two-dimensional array.

We can do addition, multiplication and other manipulations of two-dimensional arrays. In C language, we can declare arrays of any number of dimensions (i.e. 1, 2, 3 … n). We declare a n-dimensional array by putting n pair of brackets [] after the name of the array. So a three-dimensional array with values of dimensions 3, 5 and 7 respectively, will be declared as int num [3] [5] [7] ;

Example 2

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Let's have a matrix (two-dimensional array) of two rows and three columns. We want to fill it with values from the user and to display them in two rows and three columns.

Solution

To solve this problem, we use a two-dimensional array of two rows and three columns. First, we will declare the array by writing

```
int matrix [2] [3] ;
```

We declare different variables in our program. To put the values in the array, we use two nested *for* loops, which can be written as under.

```
for ( row =  0 ; row < maxrows ; row ++ )
        {
for ( col = 0 ; col < maxcols ; col ++)
{
cout << "Please enter a value for position [" << row << ", " << col << "]" ;
cin >> matrix [row] [col] ;
                }
}
```

The inner *for* loop totals the elements of the array one row at a time. It fills all the columns of a row. The outer for loop increments the row after each iteration. In the above code segment, the inner loop executes for each iteration of the outer loop. Thus, when the outer loop starts with the value of *row* 0, the inner loop is executed for a number of iterations equal to the number of columns i.e. 3 in our program. Thus the first row is completed for the three columns with positions [0,0], [0,1] and [0,2]. Then the outer loop increments the *row* variable to 1 and the inner loop is again executed which completes the second row (i.e. the positions [1,0], [1,1] and [1,2]). All the values of matrix having two rows and three columns are found.
Similarly, to display these values one by one, we again use nested loops.

Following is the code of the program.

```
//This program takes values from user to fill a two-dimensional array (matrix) having two
//rows and three columns. And then displays these values in row column format.

# include <iostream.h>
main ( )
{
        int matrix [2] [3], row, col, maxrows = 2, maxcols = 3 ;
        // get values for the matrix
for ( row = 0 ; row < maxrows ; row ++)
        {
        for (col = 0 ; col < maxcols ; col ++)
{
cout << "Please enter a value for position [" << row << ", " << col << "]  " ;
cin >> matrix [row] [col] ;
                }
}
// Display the values of matrix
cout << "The values entered for the matrix are " << endl ;
for ( row = 0 ; row < maxrows ; row ++)
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
        {
        for (col = 0 ; col < maxcols ; col ++)
{
cout << "\t" << matrix [row] [col] ;
                }
                cout << endl ;  //to start a new line for the next row
}

}
```

A sample output of the program is given below.

```
Please enter a value for position [0,0]   1
Please enter a value for position [0,1]   2
Please enter a value for position [0,2]   3
Please enter a value for position [1,0]   4
Please enter a value for position [1,1]   5
Please enter a value for position [1,2]   6

The values entered for the matrix are
                1       2       3
                4       5       6
```

vubookshoppk.com

Tips

A character array can be initialized using a string literal
Individual characters in a string stored in an array can be accessed directly using array subscript
Arrays are passed to functions by reference
To pass an array to a function, the name of the array(without any brackets) is passed along with its size
To receive an array, the function's parameter list must specify that an array will be received
Including variable names in function prototype is unnecessary. The compiler ignores these names.

Lecture No. 13

Reading Material

Deitel & Deitel – C++ How to Program — Chapter 4, 4.5, 4.9

Summary

Array Manipulation
Real World Problem and Design Recipe
Exercises

vubookshoppk.com

Array Manipulation

We have already discussed what an array is. Identical or similar values are stored in an array. The identical and similar terms here are related to the context of the problem we try to solve. For example, height or age of an individual is a number. We don't store height and age in one array as, in contextual terms, they are different things. These can not be mixed in one array. So the height of individuals will be stored in one array and the age in some other one. The idea behind the array is that whenever you have similar data with multiple values, it is easier and more elegant to store them in an array.

Let's try to find out, how to process arrays. What is the easiest way and what are the issues related to this process.

As discussed in previous lectures, whenever we come across an array, we start thinking in terms of loops. We pick up the first element of the array and process it. Then the second array element is processed and so on. Naturally that falls into an iterative structure.

Let's try to understand how to process a two dimensional array. The following example can help us comprehend it effectively.
Suppose we have a two-dimensional array of numbers. While dealing with a two-dimensional array of numbers, we should try to understand it in terms of a matrix. Matrices in mathematics have rows and column and there is always a number at each row and column intersection. Suppose we have a matrix of dimension 3 * 3 i.e. a simple two-dimensional array. We want to input some numbers to that array first. After reading these numbers, we want to output them in such a fashion that the last

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

row is printed first, followed by second last and so on till the first row that is printed at the bottom. We don't want to change the column numbers with this output. It is not a difficult task. As it is a two-dimensional array so there is a row subscript and a column subscript. Following example will make the matter further clear.
Suppose we have the following array:
int a[3][3];

We will access elements of it as: a[row index][column index] e.g. a[1][2]. This is a single element at row 1 and column 2 of array **a**.

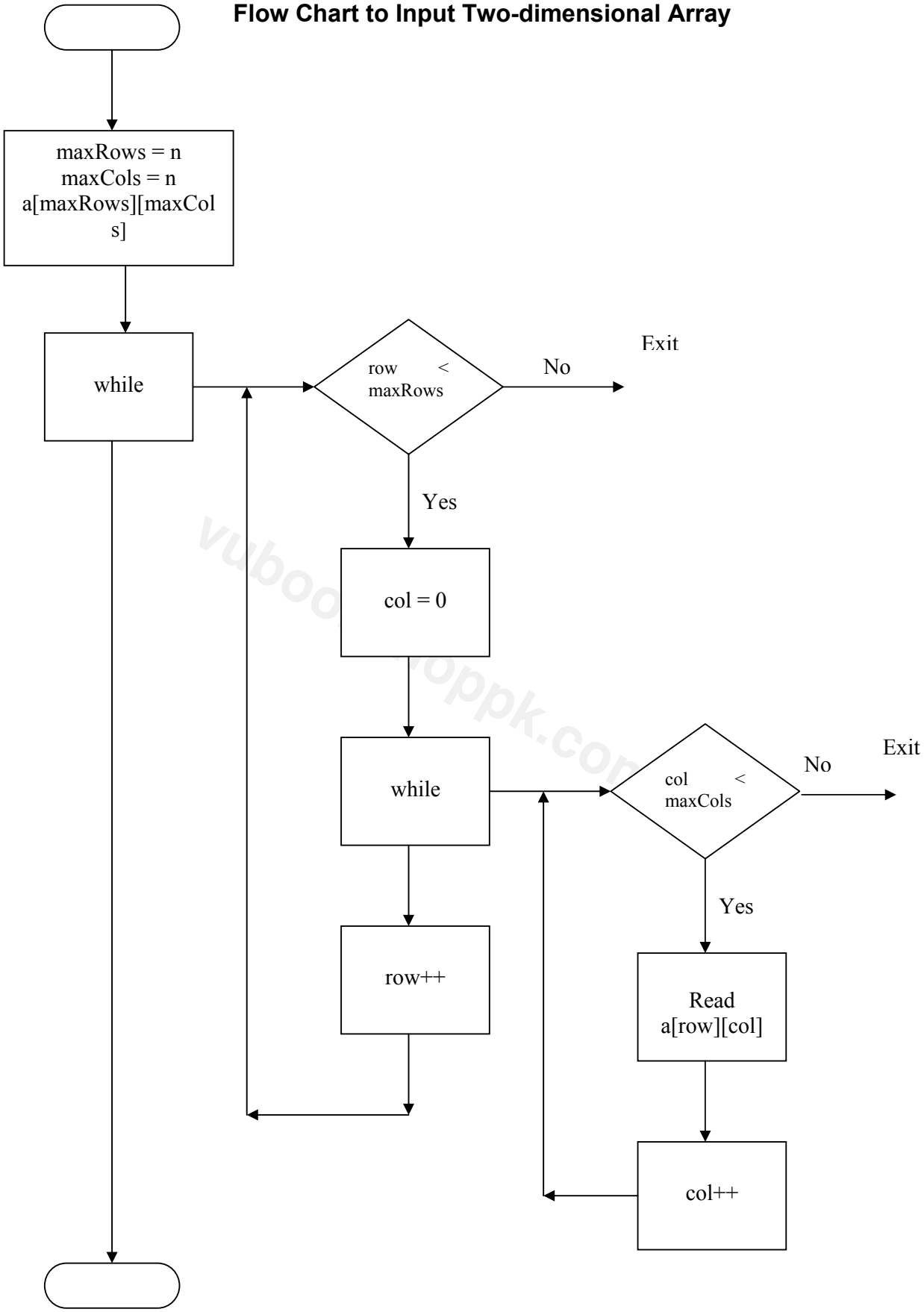
The flow chart to read in numbers into the two-dimensional array is given on the next page. See the code snippet below:

```
const int  maxRows = 3;
const int  maxCols = 3;
int row, col;

int a[maxRows][maxCols];

// To input numbers in the array
for (row = 0; row < maxRows; row ++)
{
        for(col=0; col < maxCols; col ++)
        {
                cout << "\n" << "Enter " << row << "," << col << "element: ";
                cin >> a[row][col];

        }
}
```

Now let's see what this nested loop structure is doing. The outer loop takes the first row i.e. row 0, then instantly inner loop begins which reads col 0, 1 and 2 elements of the row 0 into the array. Afterwards, control goes back to the outer loop. The row counter is incremented and becomes 1 i.e. row 1 or second row is taken for processing. Again, the inner loop reads all the elements of second row into the array. This process goes on until all the elements for three rows and three columns array are read and stored in the array called **a**.

vubookshoppk.com

Flow Chart to Input Two-dimensional Array



All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Now we want to reverse the rows of the matrix (flip the matrix) and display it. There are several ways of doing it. You might have already started thinking of how can we flip the matrix. We may declare a new matrix and copy the array elements into this matrix while flipping the elements at the same time. But we should keep in mind the problem statement. The problem statement is 'to read the array elements and then simply display it in the reverse row order'. It does not state anything about storing the elements inside the memory.

Please see the flow chart to display the flipped matrix on the next page.

Normally, we start our loops from zero and keep incrementing the counter until a certain bigger value is attained. But this is not mandatory. We can start from a bigger number and keep on decrementing the counter every time. To display the rows in reverse order, we can start from the last row and go to the first row by decrementing the row counter every time. It is very simple programming trick. However, we have to take care of the value of the index.
We can write our code inside nested loops for flipping the elements as under-

```
// To flip the elements of the matrix
cout << '\n' << "The flipped matrix is: " << '\n';

for ( row = maxRows-1; row >= 0; row --)
{
        for ( col = 0; col < maxCols; col ++)
        {
                cout << a [row][col] << '\t';
        }
        cout << '\n';
}
```

vubookshoppk.com

Note the '\t' character in the above code. It is a tab character that displays tab (spaces) at the cursor position on the screen. Similary '\n' as told in previous lectures is newline character which takes the cursor to the new line.

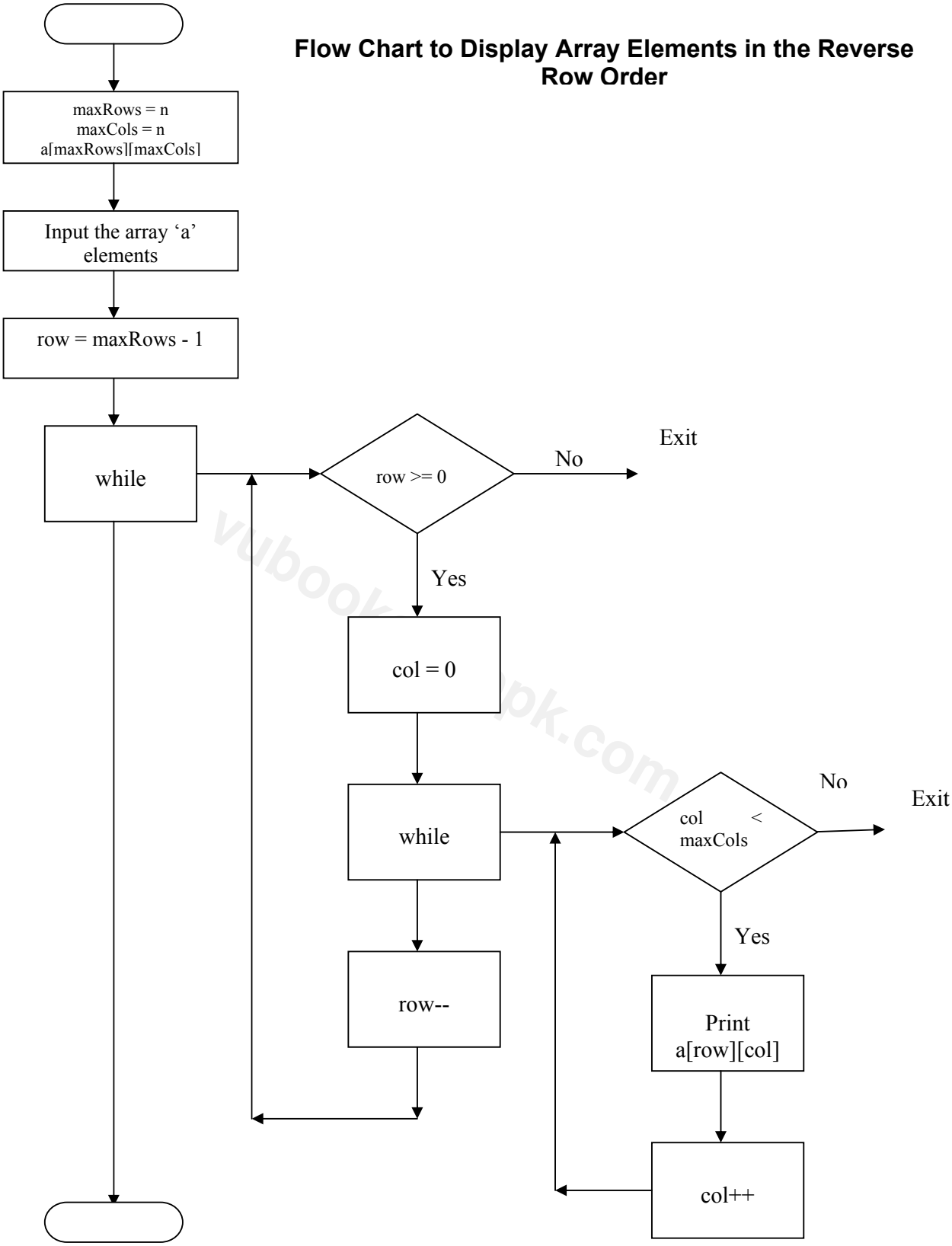
It is better to print the original matrix elements before showing the flipped matrix elements so that you can really see whether your function has flipped the matrix or not.

To run this function for the big-sized arrays, adjust the values of the maxRows and maxCols constants as the rest of the program remains the same..

Whenever we work with arrays, normally the loops are there. If the array is single dimensional, there will be one loop. A two-dimensional arrays is going to have pair of nested loops and so on.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Flow Chart to Display Array Elements in the Reverse Row Order



vubookshoppk.com

```
/*  Array Manipulation - Flipping of a Matrix (reversing the row order): This program reads a
matrix (two-dimensional array), displays its contents and also displays the flipped matrix
*/

#include <iostream.h>

const int  maxRows = 3;
const int  maxCols = 3;

void readMatrix(int arr[][maxCols]);
void displayMatrix(int a[][maxCols]);
void displayFlippedMatrix(int a[][maxCols]);

void main(void)
{

   int a[maxRows][maxCols];

   // Read the matrix elements into the array
   readMatrix(a);

   // Display the original matrix
   cout << "\n\n" << "The original matrix is: " << '\n';
   displayMatrix(a);

   // Display the flipped matrix
   cout << "\n\n" << "The flipped matrix is: " << '\n';
   displayFlippedMatrix(a);

}


void readMatrix(int arr[][maxCols])
{
   int row, col;

   for (row = 0; row < maxRows; row ++)
   {
            for(col=0; col < maxCols; col ++)
            {
                        cout << "\n" << "Enter " << row << ", " << col << " element: ";
                        cin >> arr[row][col];

            }
            cout << '\n';
   }
}

void displayMatrix(int a[][maxCols])
```

vubookshoppk.com

```
{
   int row, col;

   for (row = 0; row < maxRows; row ++)
   {

            for(col = 0; col < maxCols; col ++)
            {
                cout << a[row][col] << '\t';
            }
            cout << '\n';
    }

}

void displayFlippedMatrix(int a[][maxCols])
{
   int row, col;

   for (row = maxRows - 1; row >= 0; row --)
   {

            for(col = 0; col < maxCols; col ++)
            {
                cout << a[row][col] << '\t';
            }
            cout << '\n';
    }

}
```

vubookshoppk.com

Till now, we have only solved very simple problems to understand processing of arrays. You can test your capability of doing so through an exercise by inputting (reading in) a matrix and print it in reverse column order. Here, the rows remain the same.

Let's move on to slightly more practical problem. Before going ahead, we need to understand the concept of Transpose of a Matrix. Transpose of a matrix means that when we interchange rows and columns, the first row becomes the first column, second row becomes the second column and so on. Mathematically, the transpose can be written as:
A(i,j) should be replaced with A(j,i) where i and j are row and column indexes.

For this purpose, we take a square matrix (a matrix with equal number of rows and columns) to transpose. Here, if you are thinking in terms of loops, you are absolutely right. Let's say the array is 'a', with dimension as 'arraySize'. Please see the flow chart for this problem on the next page.

We write a pair of nested loops:

```
int temp;
for (row = 0; row < arraySize; row ++)
{

        for (col = 0; col < arraySize; col ++)
        {
                // Interchange the values here using the swapping mechanism
                temp = a[row][col];     // Save the original value in the temp variable
                a[row][col] = a[col][row];
                a[col][row] = temp;     //Take out the original value

        }
}
```

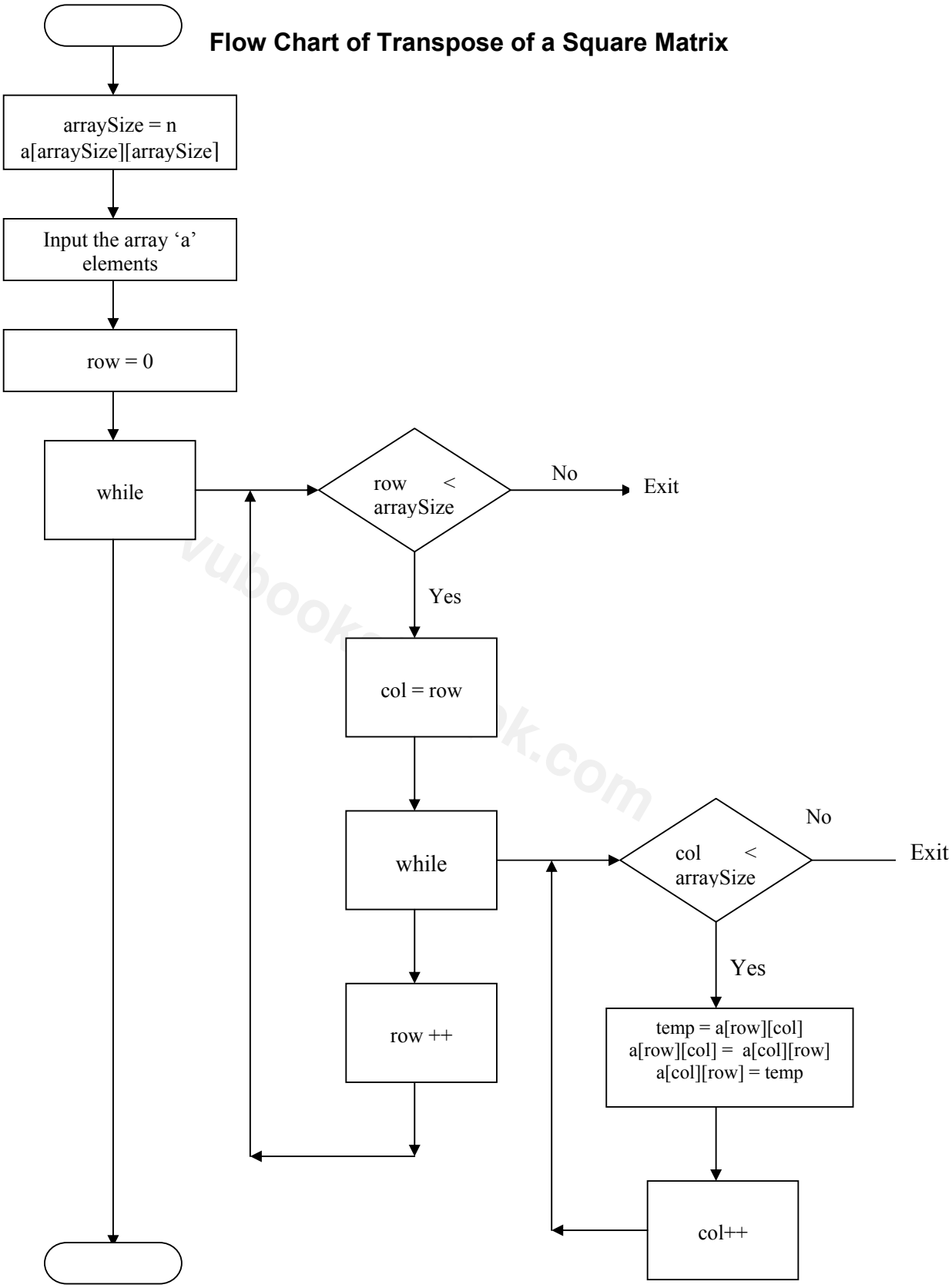
While interchanging values, we should be careful. We can't simply write: a[row][col] = a[col][row]. We will lose information this way. We need a swapping mechanism here to interchange the elements properly.

We have yet to do more to get the problem solved. You are strongly recommended to write this program and run it to see the problem area.

It is something interesting that we are interchanging the value of first row, first column with itself, which means nothing. When we are doing transpose of a matrix, the diagonal elements will remain unchanged as the row and column indexes are the same. Then we interchange the row 0, col 1 element with row 1, col 0. The row 0, col 2 element with row 2, col 0. What will happen when we process second row i.e. row 1. The row 1, col 0 will be swapped with row 0, col 1 but these are the same elements, already swapped in the above iteration. Therefore, this is the problem area that elements swapped once are swapped again to their original positions if the loops are run in all the rows and columns. As a result, the resultant matrix remains unchanged.

vubookshoppk.com

Flow Chart of Transpose of a Square Matrix



All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Then what is the solution of the problem?
Now draw a matrix on the paper and cut it diagonally. We will get two triangles i.e. upper triangle and lower triangle. We only need to interchange one triangle with the other and not the whole of the matrix. Now the question is, how we can determine the limits of triangles? By looking at a triangle, let's say upper triangle, we can see that all the rows are being processed as the triangle crosses every row. Similarly all the columns are being processed because the first row in the upper triangle covers all the columns. The only difference is that we will not process the beginning element before starting each row. That means that we will not start the inner loop (columns loop) with index 0. Rather we start with the current row number. Therefore, for first row i.e. row 0, we will process from row 0, col 0 to row 0, col arraySize-1. For second row i.e. row 1, we will process from row 1, col 1 to row 1, col arraySize-1 while in case of third row i.e. row 2, we will go from row 2, col 2 to row 2 , col arraySize-1. If you structure the loops in this manner, the correct behavior of matrix transposition will be found.

The full source code to solve this problem by taking the upper triangle and swapping it with the lower triangle is given below:

```
/*  Array Manipulation - Transpose of a Square Matrix: This program reads a matrix (two-
dimensional array), displays its contents, transposes it and then displays the transposed matrix.
*/

#include <iostream.h>

const int  arraySize = 3;

void readMatrix(int arr[][arraySize]);
void displayMatrix(int a[][arraySize]);
void transposeMatrix(int a[][arraySize]);

void main(void)
{

   int a[arraySize][arraySize];

   // Read the matrix elements into the array
   readMatrix(a);

   // Display the matrix
   cout << "\n\n" << "The original matrix is: " << '\n';
   displayMatrix(a);

   //Transpose the matrix
   transposeMatrix(a);


   //Display the transposed matrix
   cout << "\n\n" << "The transposed matrix is: " << '\n';
```

vubookshoppk.com

```
    displayMatrix(a);

}


void readMatrix(int arr[][arraySize])
{
    int row, col;

    for (row = 0; row < arraySize; row ++)
    {
            for(col=0; col < arraySize; col ++)
            {
                        cout << "\n" << "Enter " << row << ", " << col << " element: ";
                        cin >> arr[row][col];

            }
            cout << '\n';
     }
}

void displayMatrix(int a[][arraySize])
{
    int row, col;

    for (row = 0; row < arraySize; row ++)
    {

            for(col = 0; col < arraySize; col ++)
            {
                cout << a[row][col] << '\t';
            }
            cout << '\n';
    }

}

void transposeMatrix(int a[][arraySize])
{
    int row, col;
    int temp;
    for (row = 0; row < arraySize; row ++)
    {

          for (col = row; col < arraySize; col ++)
          {
              /* Interchange the values here using the swapping mechanism */

              temp = a[row][col];           // Save the original value in the temp variable
              a[row][col] = a[col][row];
```

vubookshoppk.com

```
                a[col][row] = temp;          //Take out the original value
            }
        }
    }
```

vubookshoppk.com

Real Word Problem and Design Recipe

We will take one problem that is not very complex but will follow it rigorously for all steps of design recipe.

In practical life, the employees get salaries and pay taxes honestly. Sometimes, the process of drawing salaries and payment of taxes may lead to some interesting situation. Suppose, a person draws salary of Rs. 10,000 per month. A certain percentage of tax is charged on that amount, which is deducted every month. But if the salary of the person is more than Rs. 10,000 per month, then the tax rate is different. Similarly if a person is getting Rs. 20,000 per month, he/she would be charged more under a different tax rate slab. The interesting situation develops if there is an anomaly in the tax rates i.e. a person who is getting higher salary takes home lesser money as compared to the other person with less gross salary.

To further elaborate it, we suppose that there is company 'C' where 100 or less than 100 persons are employed. The salaries of the employees and their tax rates are known to us. We are required to list those unlucky persons, who are getting lesser take-home salary (net salary) than their colleagues with less gross salaries but lower tax rates.

As per our design recipe, let's see what steps we need to follow.

A design recipe asks us to analyze the problem first and write it in a precise statement that what actual the problem is. Also by formulating the precise statement, we need to provide some examples to illustrate. At the design phase, we try to break up the problem into functional units and resort to a detailed designing. Then we move to implementation stage where the pseudo code is translated into the computer language and then the program is compiled and run to ensure that it works as expected.

At the first step i.e Analysis, we try to have a precise problem statement. Once it is established, we try to determine what are the inputs of this program. What data should be provided to this program. We will also try to determine if there are some constants required for calculation or manipulation. We list down all the constants. Then we split it up into functions and modules.

Let's try to make a precise statement of the above problem. The precise problem statement is:

"Given tax brackets and given employees gross salaries, determine those employees who actually get less take-home salary than others with lower initial income."

Suppose the tax deduction law states that:
No tax will be deducted for persons with salaries ranging from Rs. 0 to Rs. 5,000 per month or in other words tax deduction rate is 0%.
5% tax deduction will be made from the persons with salaries ranging from Rs. 5,001 to Rs. 10,000 per month.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

For persons with salaries ranging from Rs. 10,001 to Rs. 20,000, a 10% tax deduction rate would be employed.
For persons with salaries ranging from Rs. 20,001 and higher, 15% tax deduction would be made.

Taking these rules, let's formulate the problem.

Consider the example of a person with a salary of Rs. 10,000 per month. As per rules, he/she would be charged by 5% of tax rate. 5% of 10,000 is 500 rupees. So the take home salary of the person is Rs. 9500.
Now the unfortunate individual, whose gross salary is Rs, 10,001 falls in the next bracket of tax rate of 10%. He will have to pay tax worth Rs 1000.1. That means the take home salary of this person is Rs. 9000.9, which is lesser than the person with lower gross salary of Rs. 10,000. This is the problem.
We can calculate the net salaries of all individuals, determining all the unlucky ones.

Now we will carry out the analysis of the requirements. For looking into the requirements, we have to see, how to input the salaries of these people.
As stated in the problem, the number of employees of the company 'C is at most 100. So we know the size of the array. But for some other company, suppose company 'D', we don't know the number of employees. Therefore, it makes sense to take input from the user for the number of employees. Once we have determined the number of employees, we will input the gross salary of each of employees. But where will we store the gross salary? For this purpose, we will use the two-dimensional array. In the first column, we will store the gross salary. Our program after calculating the net salary for each employee will write (store) it in the second column of the array.
At the next stage, we will find out the unlucky individuals. This will be based on the analysis of algorithms. At the higher level design, we assume that there would be a way to determine the unlucky individuals. Finally, a list of unlucky employees would be prepared. For that, we will simply output the employee numbers.
We want to workout the space and storage requirements of this problem. As earlier mentioned, we will use a two dimensional array to store the gross and net salaries and output the list of unlucky employees. That means we need a storage to store that list. For this, we will take a single dimensional array of 'int' type. We will initialize the array with zero. '0' means the individual is lucky. Therefore, by default, all individuals are lucky. Whenever, we will find an unlucky individual by using the two dimensional array, we will write '1' in single dimensional array for that individual. So this is the storage requirement of the program.

Afterwards, we will discuss the interface issues. The interface guidelines are the same i.e. be polite and try to explain what is required from the user. When the program runs the user will know what is required from him/her. So there would be prompts in the program where the user will key in the data. All the input data will be coming from keyboard and displayed on the screen. This is a rudimentary interface analysis.

We have distributed the program into four major parts:
Input
Salary calculation
Identification of unlucky individuals and
Output

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Let's start the coding or detailed design phase of this program.
In a true tradition, all the four parts of the program should be function calls. The main program is very simple as it contains functions:

Get input
Calculate salary
Locate unlucky individuals
Display output

The arrays will be declared inside the main function. As we already know the maximum number of employees is 100, so we can declare it as a constant:

```
const int arraySize=100;

double sal[arraySize][2];

int lucky[arraySize] = {0};     //Notice the array initialization
```

Once this is done inside main, we want to run the input function to read the salaries of the employees. Now, inside the input data function, we will get value for number of employees from the user. We have already set the upper limit as 100 but the actual number of employees will be entered by the user of the program. If we take that input inside the input data function, what can be the problem. Well, there is no problem in taking the input within that function but the problem is the declaration of the variable 'numEmps', which contains the current number of employees. If the 'numEmps' variable is declared inside the input data function, it will be local to that function. After the input data function returns, the 'numEmps' will no longer be there because it was local to input data function and not visible in any other function. So it is better to declare the variables inside the main function. But the problem arises: how the input data function will get information about it, if we declare it inside main function. We will have to send it to input data function, either through call by reference or we can declare 'numEmp' as a global variable so that it is visible in all the functions. Global variables are useful but tricky. They exist when we need them but they exist even when we don't need them. Therefore, it might be good to declare this variable 'numEmps' inside main function and then pass by reference to the input data function.

While passing one-dimensional array to the function, we write in the function prototype as:

```
 f(int a[]);
```

However, when we pass two-dimensional array to a function, we must specify the number of columns because this depends on how a computer stores the two dimensional array in the memory. The computer stores the rows in a contiguous (row after row) fashion inside memory. Therefore , in order to locate where the first row has finished or the second row starts, it should know the number of columns. Whenever, we pass two-dimensional array to a function, the number of columns inside that array should be specified. We will pass two dimensional array 'sal' to input data function getInput() in the same manner. We also want to pass 'numEmps'

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

variable by reference using the '&' sign to this function. This will ensure that whatever the user inputs inside this function getInput(), will be available in the main function. There is another way that we get input from the user inside the main function and then pass this by value to the getInput() function. We are going to do the same in our function.

```
getInput(double sal[][2], int numEmps);
{
        for (int i = 0; i < numEmps; i ++)  //Note that this numEmps is local to this
function
        {
                cin >> sal[i][0];               // Get the gross salary for each employee
        }
}
```

To calculate tax, we will write a function. This function will be passed in similar parameters as getInput function to calculate the taxes for all the employees. There is one important point to reiterate here i.e. by default, arrays are passed by reference. That means if *getInput()* function puts some values in the 'sal' array, these are written in the 'sal' array and are available inside main function. The 'numEmps' variable on the other hand is passed by value to getInput() function. Therefore, any changes done by geInput() function will not affect the original value of 'numEmps' inside the main function.

We will continue with this problem to determine algorithm that what is the precise sequence of steps to determine the unlucky employees. For this, we need to analyze a bit more because it contains a complex 'if' condition. The function to calculate net salary also has interesting issues which will be explained in the next lecture.

Here is the source code of the first cut solution for real world problem:

```
* This is the first cut of the program to solve the real world problem of
'Unlucky Employees' */

#include <iostream.h>

void getInput(double sal[][2], int numEmps);
void calcNetSal(double sal[][2], int numEmps);
void findUnluckies(double sal[][2], int numEmps, int lucky[]);
void markIfUnlucky(double sal[][2], int numEmps, int lucky[], int upperBound, int empNbr);
void printUnluckies(int lucky[], int numEmps);

void main(void)
{
   const int arraySize=100;
   double sal[arraySize][2];
   int lucky[arraySize] = {0};
   int numEmps;

   /* Read the actual number of employees in the company */
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
    cout << "\n Please enter the total number of employees in your company: ";
    cin >> numEmps;
    cout << '\n';

    /* Read the gross salaries of the employees into the array 'sal' */
    getInput(sal, numEmps);

    /* Calculate net salaries of the employees and store them in the array */
    cout << "\n\n Calculating the net salaries ... ";
    calcNetSal(sal, numEmps);

    /* Find the unlucky employees */
    cout << "\n\n Locating the unlucky employees ... ";
    findUnluckies(sal, numEmps, lucky);

    /* Print the unlucky employee numbers */
    cout << "\n\n Printing the unlucky employees ... ";
    printUnluckies(lucky, numEmps);
}


void getInput(double sal[][2], int numEmps)
{
        for (int i = 0; i < numEmps; i++)  //Note that this numEmps is local to this function
        {
           cout << "\n Please enter the gross salary for employee no." << i << ": ";
           cin >> sal[i][0];                // Store the gross salary for each employee
        }
}

void calcNetSal(double sal[][2], int numEmps)
{
        for (int i = 0; i < numEmps; i++)  //Note that this numEmps is local to this function
        {
           if(sal[i][0] >= 0 && sal[i][0] <= 5000)
           {
              /* There is no tax deduction */
              sal[i][1] = sal[i][0];
           }
          else if(sal[i][0] >= 5001 && sal[i][0] <= 10000)
          {
               /* Tax deduction is 5% */
              sal[i][1] = sal[i][0] - (.05 * sal[i][0]);
          }
          else if (sal[i][0] >= 10001 && sal[i][0] <= 20000)
        {
             /* Tax deduction is 10% */
            sal[i][1] = sal[i][0] - (.10 * sal[i][0]);
         }
        else if (sal[i][0] >= 20001)
```

vubookshoppk.com

```
        {
            /* Tax deduction is 15% */
            sal[i][1] = sal[i][0] - (.15 * sal[i][0]);
        }
        else
        {
            /* No need to do anything here */
        }
    }
}

void findUnluckies(double sal[][2], int numEmps, int lucky[])
{
        for (int i = 0; i < numEmps; i++)  //Note that this numEmps is local to this function
        {
            if(sal[i][0] >= 0 && sal[i][0] <= 5000)
            {
                /* No need to check for unlucky employees for this tax bracket */
                ;
            }
            else if(sal[i][0] >= 5001 && sal[i][0] <= 10000)
            {
                markIfUnlucky(sal, numEmps, lucky, 5001, i);
            }
            else if (sal[i][0] >= 10001 && sal[i][0] <= 20000)
            {
                markIfUnlucky(sal, numEmps, lucky, 10001, i);
            }
            else if (sal[i][0] >= 20001)
            {
                markIfUnlucky(sal, numEmps, lucky, 20001, i);
            }
        }
}

void markIfUnlucky(double sal[][2], int numEmps, int lucky[], int upperBound, int empNbr)
{
   for (int i = 0; i < numEmps; i++)
   {
     /*
     See the if the condition below, it will mark the employee
     unlucky even if an employee in the higher tax bracket is getting
     the same amount of net salary as that of a person in the lower
     tax bracket
     */
     if (sal[i][0] < upperBound && sal[i][1] >= sal[empNbr][1])
     {
          lucky[empNbr] = 1;       //Employee marked as unlucky
          break;
     }
```

vubookshoppk.com

```
    }
}

void printUnluckies(int lucky[], int numEmps)
{
    for (int i = 0; i < numEmps; i++)
    {
        if(lucky[i] == 1)
        {
                cout <<"\n Employee No.: " << i;
        }
    }
}
```

Exercises

Suppose you have a Square matrix of order 5 * 5. Draw flow chart and write a program to input (read in) a matrix and print it in reverse column order, the rows remain the same.

Suppose you have a Square matrix of order 5 * 5. Draw flow chart and write a program to transpose the matrix, take lower triangle and swap it with upper triangle.

An Identity matrix is a square matrix whose diagonal elements are '1' and remaining elements are '0'. Suppose you are given a square matrix of size n * n. Write a program to determine if this is an Identity matrix.

vubookshoppk.com

Lecture No. 14

Reading Material

Deitel & Deitel - C++ How to Program
Chapter 5

5.1, 5.2, 5.3, 5.4, 5.5, 5.6

Summary

1) Pointers
2) Declaration of Pointers
3) Example 1 (Bubble Sort)
4) Pointers and Call By Reference
5) Example 2

vubookshoppk.com

Pointers

In the earlier lectures, we had briefly referred to the concept of pointers. Let's see what a pointer is and how it can be useful.

Pointers are a special type of variables in which a memory address is stored. They contain a memory address, not the value of the variable. The concept of the pointers can be well understood from the following example.

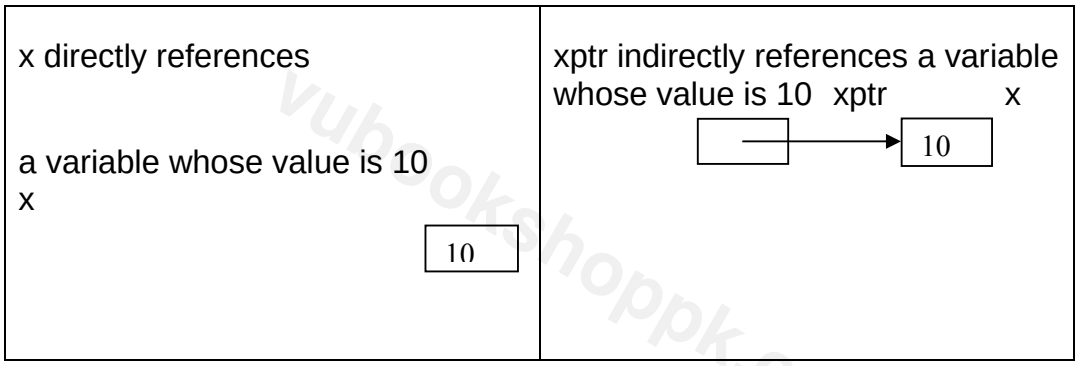
Suppose, we request someone to take a parcel to the house of a person, named Ahmad. Here the point of reference is a name. However, if we specifically tell him the number of house and the street number. Then this is a reference by the address of the house. It means that we have two ways to locate an address. To understand further the concept of memory address, the example of the computers can be helpful. In computers, one can have a name *x* which is associated with a memory location. We can have the memory address of *x*, say 6000 or whatever it is. So the simple variable names are those of specific locations in memory. But in terms of addresses, these are the addresses of those memory locations. We can use these names and addresses interchangeably to refer to memory locations. When a value is referred by a normal variable is known as direct reference. While the

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

value referred through the use of memory address may be known as indirect reference.

To understand further the terms of direct reference and indirect reference, suppose that we want to assign a value 10 to *x*. This can be done by writing *x = 10*. In this statement, the value 10 will be assigned to the memory location which has label (name) *x*. The second way to assign a value to a memory location is with reference to the address of that memory location. In other words, 'assign a value to the memory location whose address is contained in the variable (that is a pointer) on right hand side of the assignment operator'. Operators are used to refer the address of memory locations and to refer the values at those addresses.

Following figure shows directly and indirectly referencing a variable.



vubookshoppk.com

Now we will try to comprehend the concept with another daily life example. Suppose, hundreds of people are sitting in an auditorium. The host is going to announce a prize for a person amongst the audience. There are two methods to call the prizewinner to dais. The host can either call the name of the person or the number of the seat. These are equivalent to 'call by name' and 'call by address' methods. In both cases, the prize will be delivered to a person whether he is called by name or referred by address (seat number in this case). In programming, pointers are used to refer by the addresses.

Declaration of Pointers

Pointers work by pointing to a particular data type. We can have pointer to an integer, pointer to a double, pointer to a character and so on. It

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

means that a type is associated to a pointer. Pointer, being a variable, needs a name. The rules for naming a pointer are the same as for the simple variable names. The pointers are declared in a specific way. The syntax of declaring a pointer is:

```
data type *name ;
```

Here '*name*' is the name of the pointer and data type is the type of the data to which the pointer (*name*) points. There is no space between asterisk (*) and the name. Each variable being declared as a pointer must be preceded by *. The * is associated with the name of the variable, not with the data type. To associate the * (asterisk) with data type (like int*) may confuse the declaration statement. Suppose, we want to declare a pointer to an integer. We will write as:

vubookshoppk.com

```
int *myptr;
```

Here *myptr* is the name of the pointer. The easiest way to understand the pointer declaration line is the reading the statement from right to left. For the above statement, we say that *myptr* is a pointer to an integer (int). Similarly for the declaration *double *x* , *x* is a pointer to a data of type double. The declaration of *char *c* shows that *c* is a pointer to a data of type character. The declaration of multiple pointers requires the use of * with each variable name. This is evident from the following example which declares three pointers.

```
int *ptr1, *ptr2, *ptr3 ;
```

Moreover, we can mix the pointers declaration with simple variables on one line.

```
int *ptr, x, a [10] ;
```

In this declaration *ptr* is a pointer to data of type *int*, *x* is a simple variable of type *int* and *a* is an array of integers.

Whenever used, these pointers hold memory addresses.

Now we will try to understand what address a pointer holds. Suppose, we declare a pointer variable *ptr* and a variable *x* and assign a value 10 to it. We write this as under.

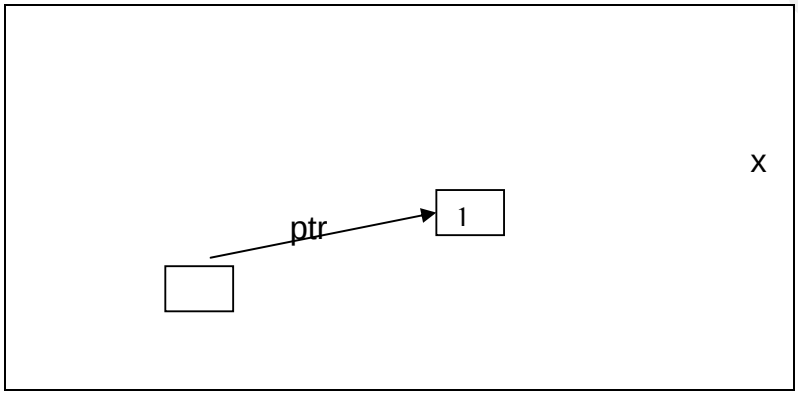
```
int *ptr ;

int x ;

x = 10 ;
```

Here *x* is a name of a memory location where a value 10 is stored. We want to store the address of this memory location (which is labeled as *x*) into the pointer *ptr*. To get the address of *x*, we use address operator i.e. &. (it is & not &&, the && is logical AND). To assign the address of *x* to pointer *ptr*, we write

```
ptr = &x ;
```

vubookshoppk.com

This statement assigns the memory address of the location *x* to the pointer *ptr*. The following figure shows a schematic representation of memory after the preceding assignment is executed.

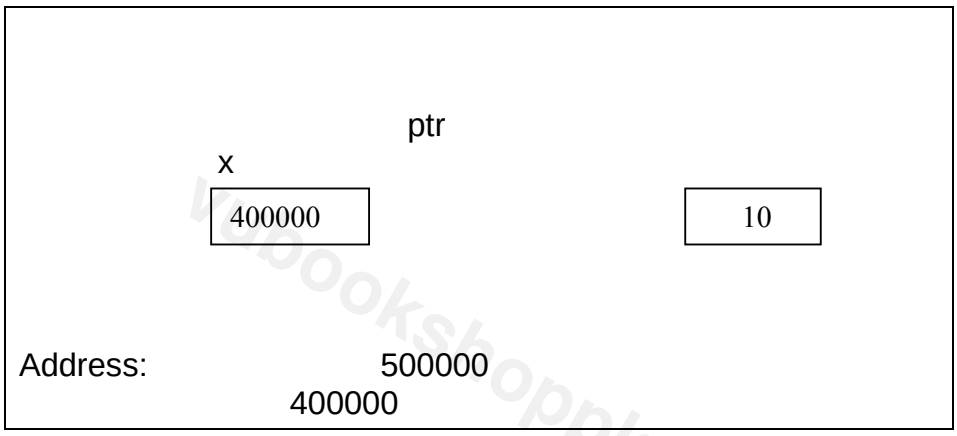


The pointers contain whole numbers as they contain memory addresses. An address can be represented only in whole numbers. Therefore, a pointer is a whole number, sufficient enough, to hold any memory address of the computer. The pointers have no specific data type.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

In the above assignment statement, we have a pointer to a memory location. Now, it can be ascertained what value is stored in that memory location. To get the value stored at a memory address, we use the dereferencing operator, represented by asterisk (*). The * is used with the name of the pointer to get the value stored at that address. To get the value stored at the memory address *ptr*, we write **ptr* which is read as the value of whatever *ptr* points to. Thus the line *z = *ptr;* means, *z* has the value of whatever *ptr* points to.

The following example can explain the representation of the pointer in memory. Assume that variable x is stored at location 400000 and pointer variable *ptr* is stored at location 500000.



vubookshoppk.com

We can use this operator (*) to get the value and can do any arithmetic operation with it. The following statements make it further clear.

```
z = *ptr + 2 ;

z = *ptr * 2 ;

z = *ptr – 2 ;
```

Here **ptr* gives the value stored at memory address where the pointer *ptr* points to.

We know that it is a good programming practice to initialize a variable when we declare it. This will ensure that there will be no unknown value in the variable at some later stage.

Similarly, we should assign an initial value to a pointer after declaring it. Taking the address of a variable and assigning it to the pointer is one way of initializing a pointer. A pointer can be initialized by assigning either value 0 or the word NULL. The NULL is a global variable declared in many header files that we include at the start of the program. The pointer initialized by NULL as *ptr = NULL;* is called null pointer which points to nothing. Similarly, when we assign a zero to a pointer like *ptr = 0*; it means that the pointer is pointing to nothing at the moment. Here zero is not considered as a valid address for a memory location. However, at some later stage, we use the pointer in an assignment statement either on left hand side to assign a value to it or as a part of an expression on right hand side. The pointer must have a valid memory address where a value should have stored. We get the address of a variable by putting & operator before the name of the variable and assign it to a pointer as in the following statement *ptr = &x;*

We know that in C language, the default mechanism of function call is 'call by value'. Sometimes we want to make a call by reference. In call by reference, we pass the address of the variable to a function by using & operator.

One of the major usages of pointers is to simulate call by reference while using it with function calls. In the calling function, we pass the address of the variable to a function being called by using & operator. We write a function call as *fn(&x)* where *&x* indicates that the address of variable *x* is being passed to the function *fn*. In the receiving function, the function must know that the parameter passed to it is an address. So the declaration of the receiving function will be as

```
void fn ( int *num)
        {
                statement(s) ;
        }
```

The *int *num* in the function declaration indicates that the receiving variable is a pointer to a memory address. In the body of the function, we will use this variable as:

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

*cin >> *num ;*

This statement describes that the value entered through the keyboard (as *cin* is used) will be stored at the memory address wherever the pointer num is pointing to.

While using value associated with the pointer, we write **num* and *&num* in case of using the address. This thing can be summarized as follows

"**num* means the value of whatever the *num* points to and

&num means the address of the variable *num*"

The pointers can appear on the left hand side exactly like ordinary variables. In this case, you would have an address statement on the right hand side. The address (operator (&)) cannot be of an expression. Rather, it is always of a simple variable. We cannot write *&(x+y)*. The address (&) would be either of *x (&x)* or of *y (&y)*. The address operator (&) operates on a simple variable. Precisely speaking, whenever we have a pointer on left hand side, the right hand side should have an address. If a pointer appears on the right hand side of an expression, it can participate in any expression. In this case, we use the operator * with the pointer name and get the value stored where the pointer points to. Obviously we can do any calculation with this value (i.e. it can be used in any expression).

Example (Bubble Sort)

You might be knowing the technique of bubble sorting. Its application helps us compare two values each time and interchange the larger and smaller values. In this way, we sort the arrays. To interchange the position of larger and smaller value, the technique of swapping is used. Swapping is very common in programming. While using this technique, we put value of one variable in a temporary location to preserve it and assign the value of second variable to the first. Then the temporary value is assigned to the second variable.

Suppose, we want to swap the values of two variables *x* and *y*. For this purpose, a third variable temp is used in the following fashion.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
temp = x ;

x = y ;

y = temp ;
```

We can write the above three statements in a program to swap the value of x and y. Now the question arises, can we call a function swap (x, y) which has a code to swap the values of x and y. We call the function swap by passing *x* and *y*. When the control comes back to the calling function, the values of *x* and *y* are the same as before. These are not swapped. This is mainly due to the fact that passing value to function swap is a call by value. It does not change the values in the calling function. The swap function receives a copy of the values and interchanges the values in that copy. The original values remain the same.

To interchange two values in a function, we make a call by reference to the function. Here comes the use of pointers. To write the swap function to interchange two values always use pointers in the function to get the swapped values in the calling function. The code fragment in our main program will be written as follows:

```
yptr = &y ;                 // address of y is stored in
yptr

xptr = &x ;                 // address of x is stored in
xptr

swap (yptr, xptr) ;  // addresses are passed
```

The receiving function must know that addresses are being passed to it. So the declaration of swap function will be:

```
swap (int *yptr, int *xptr)

{

    … … …

}
```

vubookshoppk.com

This use of pointers implements a call by reference. We can use this technique in bubble sort. The swap function can switch the elements of the array by using pointers and * operator.

The code of the program that sorts an array by bubble sort and use the swap function to interchange the elements of the array is given here.

```
/*      This program uses bubble sorting to sort a given array.
*       We use swap function to interchange the values by using pointers
*/


#include <iostream.h>
#include <stdlib.h>


/*  Prototye of function swap used to swap two values */
void swap(int *, int *) ;


main()
{
        int x [] = {1,3,5,7,9,2,4,6,8,10};
        int i, j, tmp, swaps;


        for(i = 0; i < 10; i ++)
        {
                swaps = 0;
                for(j = 0; j < 10; j ++)
                {
```

vubookshoppk.com

```
                if ( x[j] > x[j+1])      // compare two values and interchange if
needed
{
                        swaps++;
                        swap(&x[j],&x[j+1]);
                  }
                }
        //display the array's elements after each comparison
                for (j=0; j<10; j++)
                        cout << x[j] << '\t';
                cout << endl;
                if (swaps == 0)
                        break;
     }
}

void swap(int *x, int *y)    //function using pointers to interchange the values
{
    int tmp;
    if(*x > *y)
    {
         tmp = *x;
         *x = *y;
         *y = tmp;
     }
}
```

vubookshoppk.com

Following is the output of the program of bubble sort.

```
1    3    5    7    2    4    6    8
     9    10

1    3    5    2    4    6    7    8
     9    10

1    3    2    4    5    6    7    8
     9    10

1    2    3    4    5    6    7    8
     9    10

1    2    3    4    5    6    7    8
     9    10
```

vubookshoppk.com

Pointers and Call By Reference

Suppose, we have a function that performs a specific task again and again but with different variables each time. One way to do this is to pass a different variable to the function, each time, by reference. We can also write the function with pointers. In this case, before calling the function, put the address of the simple variable in the pointer variable and pass it to the function. This is a call by reference. Thus the same pointer variable can be used each time by assigning it the address of a different variable.

The mechanism behind calling a function is that, when we call a function we pass it some variables. The values of these variables are used with in the function. In call by value mechanism, the values of these variables are written somewhere else in the memory. That means a copy of these values is made. Then control goes to the called function and this copy of values is used in the function. If we have to pass a huge number of values to a function, it is not advisable to copy these huge numbers of values. In such cases, it is better to pass the reference

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

of the variables, which is a call by reference phenomenon. We perform a similar function in case of an array, where we can pass, say, 100 values (size of the array) to the called function, by only passing the name of the array. When we pass an array to a function, actually the starting address of the array is passed to the function. Thus the default calling mechanism to call a function while passing an array to it is a call by reference.

The problem with call by reference is that 'we are letting the function to change the values at their actual storage place in the memory'. Sometimes, we want to do this according to the requirement of the logic of the program. At some other occasion, we may pass the addresses for efficiency while not affecting the values at that addresses. The use of const can be helpful in overcoming this problem..

Let's look at the use of const. Consider the following line of declaration:

```
int *const myptr = &x ;
```

vubookshoppk.com

The right hand side of this assignment statement could be read as, *myptr* is a constant pointer to an integer. Whenever we use the keyword const with a variable, the value of that variable becomes constant and no other value can be assigned to it later on. We know that when we declare a constant variable like const int x ; it is necessary to assign a value to *x* and we write *const int x = 10* . After this, we cannot assign some other value to x. The value of x can not be changed as it is declared as a constant.

Now consider the previous statement

```
int *const myptr = &x ;
```

Here we declare a constant pointer to an integer. Being a constant pointer, it should immediately point to something. Therefore, we assign this pointer an address of a variable *x* at the time of declaration. Now this pointer cannot be changed. The pointer *myptr* will hold the address of variable *x* throughout the program. This way, it becomes just another name for the variable *x*. The use of constant pointers is not much useful.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

The use of keyword const in declaration statement is a little tricky. The statement

```
int *const myptr = &x ;
```

means *myptr* is a constant pointer to an integer. But if we change the place of const in this statement and write

```
const int *myptr = &x ;
```

This statement describes that *myptr* is a pointer to a constant integer. This means that the value of pointer *myptr* can be changed but the value stored at that location cannot be changed. This declaration is useful. It has a common use in call by reference mechanism. When we want to pass the arguments to a function by reference without changing the values stored at that addresses. Then we use this construct of declaration (i.e. const int *myptr) in the called function declaration. We write the declaration of the function like

```
fn ( const int *myptr)
    {
        ….
    }
```

This declaration informs the function that the receiving value is a constant integer. The function cannot change this value. Thus we can use the address of that value for manipulations but cannot change the value stored at that location.

Example 2

Let's consider an example in which we use the pointers to make a call by reference. We want to convert the lowercase letters of a string (character array), to their corresponding uppercase letters.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

We write a function convertToUppercase, which processes the string s one character at a time using pointer arithmetic. In the body of the function, we pass the character to a function islower. This function returns true if the character is a lowercase letter and false otherwise. The characters in the range 'a' through 'z' are converted to their corresponding uppercase letters by function toupper. Function toupper takes one character as an argument. If the character is a lowercase letter, the corresponding uppercase letter is returned, otherwise the original character is returned. The functions toupper and islower are part of the character handling library <ctype.h>. So we have to include this header file in our program. We include it in the same way, as we include <iostream.h>.

The complete code of the program is given below.

```
//This program converts a string into an uppercase string

# include <iostream.h>
# include <ctype.h>
# include <stdlib.h>

//declare the functions prototype
void convertToUppercase (char *)
main ()
{
        char s [30] = "Welcome To Virtual University" ;
        cout << "The string before conversion is:   "  << s << endl ;
convertToUppercase ( s) ;      //function call
        cout << "The string after conversion is:  "  << s ;
}

void convertToUppercase (char *sptr)
{
        while ( *sptr != '\0' )
        {
        if ( islower ( *sptr) )
                *sptr = toupper ( *sptr );        //convert to uppercase
        ++ sptr;                                                    // move sptr to the next
character
}
}
```

Following is the output of the program.

```
The string before conversion is :       Welcome To Virtual University
The string after conversion is :        WELCOME TO VIRTUAL UNIVERSITY
```

Exercise

1. Modify the above program so that it gets a string from user and converts it into lowercase.

vubookshoppk.com

2. Write a program, which converts a string of uppercase letters into its corresponding lowercase letters string.

vubookshoppk.com

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Lecture No. 15

Reading Material

Deitel & Deitel - C++ How to Program — Chapter 5, 5.7, 5.8

Summary

6) Introduction
7) Relationship between Pointers and Arrays
8) Pointer Expressions and Arithmetic
9) Pointers Comparison
10) Pointer, String and Arrays
11) Tips

vubookshoppk.com

Introduction

In the previous lecture, we had just started the discussion on the topic of pointers. This topic is little complicated, yet the power we get with the pointers is very interesting. We can do many interesting things with pointers. When other languages like Java evolve with the passage of time, pointers are explicitly excluded. In today's lecture, we will discuss pointers, the relationship between pointers and arrays, pointer expressions, arithmetic with pointers, relationship between arrays and pointer, strings etc.

Relationship between Pointers and Arrays

When we write *int x,* it means that we have attached a symbolic name *x,* at some memory location. Now we can use *x = 10* which replaces the value at that memory location with 10. Similarly while talking about arrays, suppose an array as *int y[10]*. This means that we have reserved memory spaces for ten integers and named it collectively as *y*. Now we will see what actually y is? '*y*' represents the memory address of the beginning of this collective memory space. The first element of the array can be accessed as *y[0]*. Remember arrays index starts from 0 in C language, so the memory address of first element i.e. *y[0]* is stored in *y*.

> "The name of the array is a constant pointer which contains the memory address of the first element of the array"

The difference between this and an ordinary pointer is that the array name is a constant pointer. It means that the array name will always point to the start of the array. In other words, it always contains the memory address of the first element of

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

the array and cannot be reassigned any other address. Let's elaborate the point with the help of following example.

```
int y[10];
int *yptr;
```

In the above statements, we declare an array *y* of ten integers and a pointer to an integer i.e. *yptr*. This pointer may contain a memory address of an integer.

```
yptr = y;
```

This is an assignment statement. The value of *y i.e.* the address of the first element of the array is assigned to *yptr*. Now we have two things pointing to the same place, *y* and *yptr.* Both are pointing to the first element of the array. However, *y* is a constant pointer and always points to the same location whereas *yptr* is a pointer variable that can also point to any other memory address.

Pointer Expressions and Arithmetic

Suppose we have an array *y* and *yptr*, a pointer to array. We can manipulate arrays with both *y* and *yptr*. To access the fourth element of the array using *y,* we can say *y[3];* with *yptr,* we can write as **(yptr + 4).* Now we have to see what happens when we increment or add something to a pointer. We know that *y* is a constant pointer and it can not be incremented. We can write *y[0], y[1]* etc. On the other hand, *yptr* is a pointer variable and can be written as the statement *yptr = y.* It means that *yptr* contains the address of the first element of the array. However, when we say *yptr++,* the value of *yptr* is incremented. But how much? To explain it further, we increment a normal integer variable like *x++. If x* contains 10, it will be incremented by 1 and become 11. The increment of a pointer depends on its data type. The data type, the pointer points to, determines the amount of increment. In this case, *yptr* is an integer pointer. Therefore, when we increment the *yptr,* it points to the next integer in the memory. If an integer occupies four bytes in the memory, then the *yptr++;* will increment its value by four. This can be understood from the following example.

```
// This program will print the memory address of a pointer and its incremented address.

#include<iostream.h>

main()
{
        int y[10];      // an array of 10 integers
        int *yptr;      // an integer pointer
        yptr = y;       // assigning the start of array address to pointer

        // printing the memory address
        cout << "The memory address of yptr = " << yptr << endl ;
        yptr++;         // incrementing the pointer

        // printing the incremented memory address
        cout << "The memory address after incrementing yptr = " << yptr << endl;
}
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

In the above program, the statement *cout << yptr* will show the memory address the *yptr* points to. You will notice the difference between the two printed addresses. By default, the memory address is printed in hexadecimal by the C output system. Therefore, the printed address will be in hexadecimal notation. The difference between the two addresses will be four as integer occupies four bytes and *yptr* is a pointer to an integer.

> "When a pointer is incremented, it actually jumps the number of memory spaces according to the data type that it points to"

The sample out put of the program is:

```
The memory address of yptr = 0x22ff50
The memory address after incrementing yptr = 0x22ff54
```

yptr which was pointing to the start of the array *y*, starts pointing to the next integer in memory after incrementing it. In other words, y*ptr* is pointing to the 2nd element of the array. On being incremented again, the *yptr* will be pointing to the next element of the array i.e. *y[2*], and so on. We know that & is address operator which can be used to get the memory address. Therefore, we can also get the address of the first element of the array in *yptr* as:

```
yptr = &y[0] ;
```

y[0] is a single element and its address can be got with the use of. the address operator (&). Similarly we can get the address of 2nd or 3rd element as *&y[1], &y[2]* respectfully. We can get the address of any array element and assign it to *yptr*.

Suppose the *yptr* is pointing to the first element of the array *y*. What will happen if we increment it too much? Say, the array size is 10. Can we increment the *yptr* up to 12 times? And what will happen? Obviously, we can increment it up to 12 times. In this case, y*ptr* will be pointing to some memory location containing garbage (i.e. there may be some value but is useless for us). To display the contents where the *yptr* is pointing we can use *cout* with dereference pointer as:

```
cout << *yptr ;
```

The above statement will display the contents where *yptr* is pointing. If the yptr is pointing to the first element of the array, *cout << *yptr* will display the contents of the first element of the array (i.e. y[0]). While incrementing the *yptr* as *yptr ++*, the statement *cout << * yptr* will display the contents of the 2nd element of the array(i.e. y[1]) and so on.

Here is an example describing different methods to access array elements.

```
/* This program contains different ways to access array elements */
```

vubookshoppk.com

```
#include <iostream.h>

main ()
{
        int y[10] = {0,5,10,15,20,25,30,35,40,45};
        int *yptr;

        yptr = y; // Assigning the address of first element of array.

        cout << "Accessing 6th element of array as y[5] = " << y[5] << endl;

        cout  <<  "Accessing 6th element of array as *(yptr + 5) = " << *(yptr + 5) << endl;

        cout << "Accessing 6th element of array as yptr[5] = " << yptr[5] << endl;
}
```

The output of the program is:

```
Accessing 6th element of array as y[5] = 25
Accessing 6th element of array as *(yptr + 5) = 25
Accessing 6th element of array as yptr[5] = 25
```

In the above example, there are two new expressions i.e. **(yptr+5)* and *yptr[5]*. In the statement **(yptr+5)*, *yptr* is incremented first by 5 (parenthesis are must here). Resultantly, it points to the 6th element of the array. The dereference pointer gives the value at that address. As *yptr* is a pointer to an integer, so it can be used as array name. So the expression *yptr[5]* gives us the 6th element of the array.

The following example can explain how we can step through an entire array using pointer.

```
/* This program steps through an array using pointer */

#include <iostream.h>

main ()
{
        int y[10] = {10,20,30,40,50,60,70,80,90,100};
        int *yptr, i;

        yptr = y; // Assigning the address of first element of array.

        for (i = 0; i < 10 ; i ++)
        {
                cout  << "\n The value of the element at position " << i << " is " << *yptr;
                yptr ++ ;
        }
}
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

The output of the program is:

```
The value of the element at position 0 is 10
The value of the element at position 1 is 20
The value of the element at position 2 is 30
The value of the element at position 3 is 40
The value of the element at position 4 is 50
The value of the element at position 5 is 60
The value of the element at position 6 is 70
The value of the element at position 7 is 80
The value of the element at position 8 is 90
The value of the element at position 9 is 100
```

Consider another example to elaborate the pointer arithmetic.

```
/* Program using pointer arithmetic */

#include <iostream.h>

main()
{
        int x =10;
        int *yptr;

        yptr = &x;

        cout << "The address yptr points to = " << yptr << endl ;
        cout << "The contents yptr  points to = " << *yptr << endl;

        (*yptr) ++;

        cout << "After increment, the contents are  " << *yptr << endl;
        cout << "The value of x is =  " << x << endl;
}
```

vubookshoppk.com

The output of the program is:

```
The address yptr points to = 0x22ff7c
The contents yptr  points to = 10
After increment, the contents are  11
The value of x is =  11
```

Here the statement *(*yptr)* ++ is read as "increment whatever *yptr* points to". This will increment the value of the variable. As *yptr* and *x* both are pointing to the same location, the contents at that location becomes 11. Consider the statement **yptr + 3 ;* This is an expression and there is no assignment so the value of x will not be changed

where as the statement **yptr += 3;* will increment the value of *x* by 3. If we want to increment the pointer and not the contents where it points to, we can do this as *yptr ++;* Now where *yptr* is pointing? The *yptr* will be now pointing four bytes away from the memory location of *x*. The memory location of *x* is a part of program, yet after incrementing *yptr*, it is pointing to some memory area, which is not part of the program. Take this as an exercise. Print the value of *yptr* and **yptr* and see what is displayed? Be sure, it is not illegal and the compiler does not complain. The error will be displayed if we try to write some value at that memory address.

> "When a pointer is used to hold the memory address of a simple variable, do not increment or decrement the pointer. When a pointer is used to hold the address of an array, it makes sense to increment or decrement the pointer "

Be careful while using pointers, as no warning will be given, in case of any problem. As pointers can point at any memory location, so one can easily get the computers crashed by using pointers.

Remember that incrementing the pointer and incrementing the value where the pointer points to are two different things. When we want to increment the pointer, to make it point to next element in the memory, we write as *(yptr++);* Use parenthesis when incrementing the address. If we want to increment the value where the pointer points to, it can be written as *(*yptr) ++;* Keep in mind the precedence of operator. Write a program to test this.

The decrement of the pointer is also the same. *yptr --*; *yptr -= 3 ;* will decrement the *yptr*. Whereas the statement *(*yptr) --;* will decrement the value where the *yptr* is pointing. So if the *yptr* is pointing to *x* the value of *x* will be decremented by 1.

Pointers are associated to some data type as pointer to integer, pointer to float and pointer to char etc. When a pointer is incremented or decremented, it changes the address by the number of bytes occupied by the data type that the pointer points to. For example, if we have a pointer to an integer, by incrementing the pointer the address will be incremented by four bytes, provided the integer occupies four bytes on that machine. If it is a pointer to float and float occupies eight bytes, then by incrementing this pointer, its address will be incremented by eight bytes. Similarly, in case of a pointer to a char, which normally takes one byte, incrementing a pointer to char will change the address by one. If we move to some other architecture like Macintosh, write a simple program to check how many bytes integer, float or char is taking with the use of simple pointer arithmetic. In the modern operating systems like windows XP, windows 2000, calculator is provided under tools menu. Under the view option, select scientific view. Here we can do hexadecimal calculations. So we can key in the addresses our programs are displaying on the screen and by subtracting, we can see the difference between the two addresses. Try to write different programs and experiment with these.

We have seen that we can do different arithmetic operations with pointers. Let's see can two pointers be added? Suppose we have two pointers *yptr1* and *yptr2* to integer and written as *yptr1 + yptr2 ;* The compiler will show an error in this statement. Think logically what we can obtain by adding the two memory addresses. Therefore, normally compiler will not allow this operation. Can we subtract the pointers? Yes,

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

we can. Suppose we have two pointers pointing to the same memory address. When we subtract these, the answer will be zero. Similarly, if a pointer is pointing to the first element of an integer array while another pointer pointing to the second element of the array. We can subtract the first pointer from second one. Here the answer will be one, i.e. how many array elements are these two pointers apart.

Consider the following sample program:

```
/* Program using the pointer subtraction */

#include <iostream.h>

main ()
{
        int y[10], *yptr1, *yptr2;

        yptr1 = &y[0];
        yptr2 = &y[3];

        cout << " The difference = " << yptr2 - yptr1;
}
```

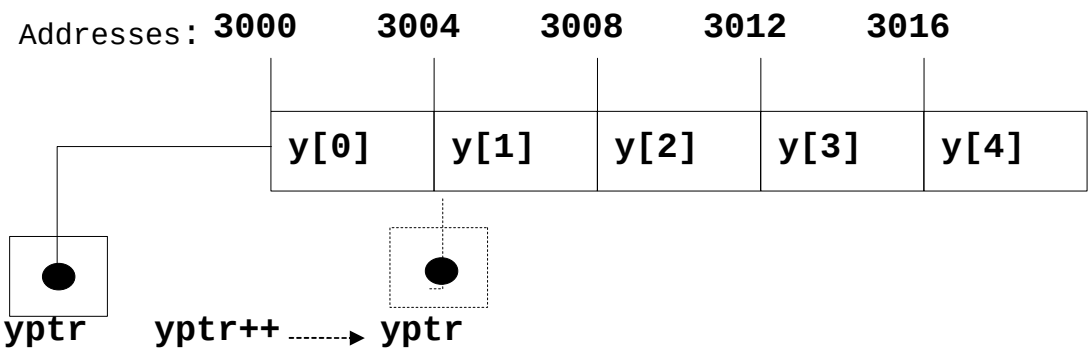
The output of the program is:

```
The difference = 3
```

In the above program, we have taken two integer pointers *yptr1* and *yptr2* and an integer array *y[10]*. The pointer *yptr1* is pointing to the address of the first element of the array while *yptr2* is pointing to the 4th element of the array. The difference between these two pointers can be shown by using *cout* statement. Here the result should be twelve. But the program will show the result as three. When we increment an integer pointer by 1, we have seen that the address is changed by four. When we subtract pointers, it tells us the distance between the two elements that the pointers pointed to. It will tell us how many array elements are between these two pointers. As the *yptr1* is pointing to *y[0]* and the *yptr2* is pointing to *y[3]*, so the answer is three. In a way, it tells how many units of data type (pointers data type) are between the two pointers. Pointer addition is not allowed, however, pointer subtraction is allowed as it gives the distance between the two pointers in units, which are the same as the data type of the pointer.

A memory image of an array with a pointer.

vubookshoppk.com



This diagram shows how an array occupies space in the memory. Suppose, we have an integer array named *y* and *yptr* is a pointer to an integer and is assigned the address of the first element of the array. As this is an integer array, so the difference between each element of the array is of four bytes. When the *yptr* is incremented, it starts pointing to the next element in the array.

Pointer Comparison

We have seen pointers in different expressions and arithmetic operations. Can we compare pointers? Yes, two pointers can be compared. Pointers can be used in conditional statements as usual variables. All the comparison operators can be used with pointers i.e. less than, greater than, equal to, etc. Suppose in sorting an array we are using two pointers. To test which pointer is at higher address, we can compare them and take decision depending on the result.

Again consider the two pointers to integer i.e. *yptr1* and *yptr2*. Can we compare **yptr1* and **yptr2*? Obviously **yptr1* and **yptr2* are simple values. It is the value of integer *yptr1*, *yptr2* points to. When we say *yptr1 > *yptr2, this is a comparison of simple two integer values. Whenever we are using the dereference pointer (pointers with *), all normal arithmetic and manipulation is valid. Whenever we are using pointers themselves, then certain type of operations are allowed and restrictions on other. Make a list what can we do with a pointer and what we cannot.

Consider a sample program as follows:

```
/* Program using the dereference pointer comparison */

#include <iostream.h>

main ()
{
        int x, y, *xptr, *yptr;

        cout << " \n Please enter the value of x = " ;
        cin >> x ;
```

vubookshoppk.com

```
        cout << " \n Please enter the value of y = ";
        cin >> y ;

        xptr = &x;
        yptr = &y;

        if (*xptr > *yptr )
        {
                cout << " \n x is greater than y ";
        }
        else
        {
                cout << "\n y is greater than x ";
        }
}
```

The output of the program is;

```
Please enter the value of x = 6

Please enter the value of y = 9

y is greater than x
```

vubookshoppk.com

Pointer, String and Arrays

We have four basic data types i.e. char, int, float and double. Character strings are arrays of characters. Suppose, there is a word or name like Amir to store in one entity. We cannot store it into a char variable because it can store only one character. For this purpose, a character array is used. We can write it as:

```
char name [20];
```

We have declared an array *name* of 20 characters .It can be initialized as:

```
name[0] = 'A' ;
name[1] = 'm' ;
name[2] = 'i' ;
name[3] = 'r' ;
```

Each array element is initialized with a single character enclosed in single quote. We cannot use more than one character in single quotes, as it is a syntax error. Is the initialization of the array complete? No, the character strings are always terminated by null character '\0'. Therefore, we have to put the null character in the end of the array.

name[4] = '\0' ;

Here we are using two characters in single quotes. But it is a special case. Whenever back slash (\ ) is used, the compiler considers both the characters as single (also known as escape characters). So '\n' is new line character, '\t' a tab character and '\0' a null character. All of these are considered as single characters. What is the benefit of having this null character at the end of the string? Write a program, do not use the null character in the string and try to print the character array using *cout* and see what happens? *cout* uses the null character as the string terminating point. So if *cout* does not find the null character it will keep on printing. Remember, if we want to store fifteen characters in an array, the array size should be at least sixteen i.e. fifteen for the data and one for the null character. Do we always need to write the null character at the end of the char array by ourselves? Not always, there is a short hand provided in C, i.e. while declaring we can initialize the arrays as:

char name[20] = "Amir";

When we use double quotes to initialize the character array, the compiler appends null character at the end of the string.

"Arrays must be at least one character space larger than the number of printable characters which are to be stored"

Example:

Write a program which copies a character array into given array.

Solution:

Here is the complete code of the program:

```
/* This program copies a character array into a given array */

#include <iostream.h>

main( )
{
        char strA[80] = "A test string";
        char strB[80];

        char *ptrA;      /* a pointer to type character */
        char *ptrB;      /* another pointer to type character */

        ptrA = strA;    /* point ptrA at string A */
        ptrB = strB;    /* point ptrB at string B */

        while(*ptrA != '\0')
        {
                *ptrB++ = *ptrA++;   // copying character by character
        }

        *ptrB = '\0';
```

vubookshoppk.com

```
        cout << "String in strA = " << strA << endl; /* show strA on screen */
        cout << "String in strB = " << strB << endl;  /* show strB on screen */
}
```

The output of the program is:

```
String in strA = A test string
String in strB = A test string
```

Explanation:

Suppose, we have declared a char array named *strA* of size 80 and initialized it with some value say "A test String" using the double quotes. Here we don't need to put a null character. The compiler will automatically insert it. But while declaring another array *strB* of the same size, we declare two char pointers **ptrA* and **ptrB*. The objective of this exercise is to copy one array into another array. We have assigned the starting address of array *strA* to *ptrA* and *strB* to *ptrB*. Now we have to run a loop to copy all the characters from one array to other. To terminate the loop, we have to know about the actual number of characters or have to use the string termination character. As we know, null character is used to terminate a string, so we are using the condition in 'while loop' as: **ptrA != '\0'*, simply checking that whatever *ptrA* is pointing to is not equal to '\0'. Look at the statement **ptrB++ = *ptrA++*. What has happened in this statement? First of all, whatever *ptrA* is pointing to will be assigned to the location where *ptrB* is pointing to. When the loop starts, these pointers are pointing to the start of the array. So the first character of *strA* will be copied to the first character of *strB*. Afterwards, the pointers will be incremented, not the values they are pointing to. Therefore, *ptrA* is pointing to the 2nd element of the array *strA* and *ptrB* is pointing to the 2nd element of the array *strB*. In the 2nd repetition, the loop condition will be tested. If *ptrA* is not pointing to a null character the assignment for the 2nd element of the array takes place and so on till the null character is reached. So all the characters of array *strA* are copied to array *strB*. Is this program complete? No, the array *strB* is not containing the null character at the end of the string. Therefore, we have explicitly assigned the null character to strB. Do we need to increment the array pointer? No, simply due to the fact that in the assignment statement (*ptrA++ = *ptrB++;), the pointers are incremented after the assignment. This program now successfully copies one string to other using only pointers. We can also write a function for the string copy. The prototype of the function will be as:

```
void myStringCopy (char *destination, const char *source) ;
```

This function takes two arguments. The first one is a pointer to a char while second argument is a const pointer to char. The destination array will be changed and all the characters from source array are copied to destination. At the same time, we do not want that the contents of source should be changed. So we used the keyword const with it. The keyword const makes it read only and it can not be changed accidentally.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

If we try to change the contents of source array, the compiler will give an error. The body is same, as we have seen in the above program.
This function will not return anything as we are using pointers. It is automatically call by reference. Whenever arrays are passed to functions, a reference of the original array is passed. Therefore, any change in the array elements in the function will change the actual array. The values will be written to the original array. If these are simple variables, we will have to send the address and get the called program to change it. Therefore, we do not need to return anything from this function after successfully copying an array into the other.

Here is the code of the function. Write a program to test this function.

```
void myStringCopy (char *destination, const char *source)
{
        while(*source != '\0')
        {
                *destination++ = *source++;
        }
        *destination = '\0';
}
```

We can also write the string copy function using arrays. Here is the code of the myStringCopy function using arrays notation.

```
void myStringCopy(char dest[], char source[])
 {
    int i = 0;

    while (source[i] != '\0')
    {
      dest[i] = source[i];
      i++;
    }
    dest[i] = '\0';
 }
```

Exercise:

1) Print out the address and the value of a character pointer pointing to some character.
2) Write a function which copies an array of integers from one array to other

Tips

- While incrementing the pointers, use the parenthesis
- Increment and decrement the pointers while using arrays
- When a pointer is incremented or decremented, it changes the address by the

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

number of bytes occupied by the data type that the pointer points to

- Use key word const with pointers to avoid unwanted changes
- The name of array is a constant pointer. It cannot be reassigned

vubookshoppk.com

Lecture No. 16

Reading Material

Deitel & Deitel - C++ How to Program	Chapter 5, 18 5.9, 5.10, 18.4

Summary

- Pointers (continued)
- Multi-dimensional Arrays
- Pointers to Pointers
- Command-line Arguments
- Exercises
- Tips

vubookshoppk.com

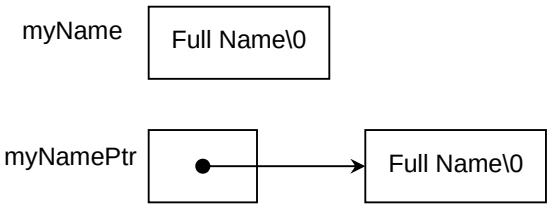
Pointers (continued)

We will continue with the elaboration of the concept of pointers in this lecture. To further understand pointers, let's consider the following statement.

```
char myName[] = "Full Name";
```

This statement creates a 'char' type array and populates it with a string. Remember the character strings are null ('\0') terminated. We can achieve the same thing with the use of pointer as under:

```
char * myNamePtr = "Full Name";
```



Let's see what's the difference between these two approaches?
When we create an array, the array name, 'myName' in this case, is a constant pointer. The starting address of the memory allocated to string "FullName" becomes the

contents of the array name 'myName' and the array name 'myName' can not be assigned any other value. In other words, the location to which array names points to can not be changed. In the second statement, the 'myNamePtr' is a pointer to a string "FullName", which can always be changed to point to some other string.

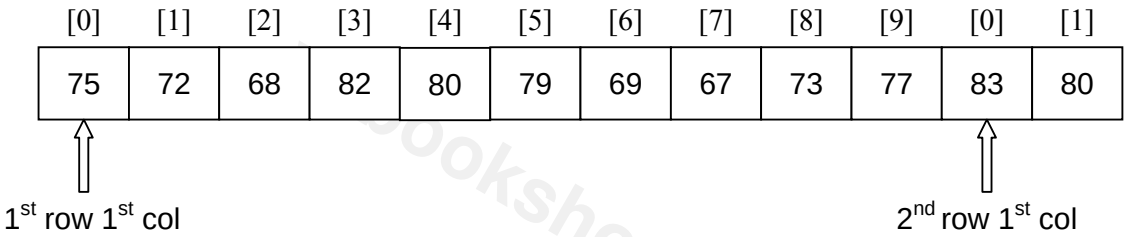
Hence, the array names can be used as pointers but only as constant ones.

Multi-dimensional Arrays

Now we will see what is the relationship between the name of the array and the pointer. Suppose we have a two-dimensional array:

```
char multi[5][10];
```

In the above statement, we have declared a 'char' type array of 5 rows and 10 columns.



Multi-dimensional array in the memory

As discussed above, the array name points to the starting memory location of the memory allocated for the array elements. Here the question arises where the 'multi' will be pointing if we add 1 to 'multi'.

We know that a pointer is incremented by its type number of bytes. In this case, 'multi' is an array of 'char' type that takes 1 byte. Therefore, 'muti+1' should take us to the second element of the first row (row 0). But this time, it is behaving differently. It is pointing to the first element (col 0) of the second row (row 1). So by adding '1' in the array name, it has jumped the whole row or jumped over as many memory locations as number of columns in the array. The width of the columns depends upon the type of the data inside columns. Here, the data type is 'char', which is of 1 byte. As the number of columns for this array 'multi' is 10, it has jumped 10 bytes.

Remember, whenever some number is added in an array name, it will jump as many rows as the added number. If we want to go to the second row (row 1) and third column (col 2) using the same technique, it is given ahead but it is not as that straight forward. Remember, if the array is to be accessed in random order, then the pointer approach may not be better than array indexing.

We already know how to dereference array elements using indexing. So the element at second row and third column can be accessed as 'multi[1][2]'.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

To do dereferencing using pointers we use '*' operator. In case of one-dimensional array, '*multi' means 'the value at the address, pointed to by the name of the array'. But for two-dimensional array '*multi' still contains an address of the first element of the first row of the array or starting address of the array 'multi'. See the code snippet to prove it.

```
/* This program uses the multi-dimensional array name as pointer */

#include <iostream.h>

void main(void)
{
        //To avoid any confusion, we have used ‘int’ type below
        int multi[5][10];
        cout << "\n The value of multi is: " << multi;

        cout << "\n The value of *multi is: " << *multi;
}
```

Now, look at the output below:

```
The value of multi is: 0x22feb0
The value of *multi is: 0x22feb0
```

It is pertinent to note that in the above code, the array ‘multi’ has been changed to ‘int’ from ‘char’ type to avoid any confusion.

To access the elements of the two-dimensional array, we do double dereferencing like '**multi'. If we want to go to, say, 4th row (row 3), it is achieved as 'multi + 3' . Once reached in the desired row, we can dereference to go to the desired column. Let's say we want to go to the 4th column (col 3). It can be done in the following manner.

```
*(*(multi+3)+3)
```

This is an alternative way of manipulating arrays. So 'multi[3][3]' element can also be accessed by '*(*(multi+3)+3)'.

There is another alternative of doing this by using the normal pointer. Following code reflects it.

```
/* This program uses array manipulation using indexing */

#include <iostream.h>

void main(void)
{
   int multi [5][10];
   int *ptr;              // A normal ‘int’ pointer
```

vubookshoppk.com

```
    ptr = *multi;          // 'ptr' is assigned the starting address of the first row

    /* Initialize the array elements */
    for(int i=0; i < 5; i++)
    {
       for (int j=0; j < 10; j++)
       {
            multi[i][j] = i * j;
       }
    }

    /* Array manipulation using indexing */
    cout << "\n Array manipulated using indexing is: \n";
    for(int i=0; i < 5; i++)
    {
       for (int j=0; j < 10; j++)
       {
            cout << multi[i][j] << '\t';
       }
       cout << '\n';
    }

    /* Array manipulation using pointer */
    cout << "\n Array manipulated using pointer is: \n";
    for(int k=0; k < 50; k++, ptr ++)          // 5 * 10 = 50
    {
       cout << *ptr << '\t';
    }
}
```

The output of this program is:

```
Array manipulated using indexing is:
0    0    0    0    0    0    0    0    0    0
0    1    2    3    4    5    6    7    8    9
0    2    4    6    8    10   12   14   16   18
0    3    6    9    12   15   18   21   24   27
0    4    8    12   16   20   24   28   32   36

 Array manipulated using pointer is:
0    0    0    0    0    0    0    0    0    0    0    1    2    3    4    5
6    7    8    9    0    2    4    6    8    10   12   14   16   18   0    3
6    9    12   15   18   21   24   27   0    4    8    12   16   20   24
28   32   36
```

The above line of output of array manipulation is wrapped because of the fixed width of the table. Actually, it is a single line.

Why it is a single line? As discussed in the previous lectures, computer stores array in straight line (contiguous memory locations). This straight line is just due to the fact that a function accepting a multi-dimensional array as an argument, needs to know all the dimensions of the array except the leftmost one. In case of two-dimensional array,

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

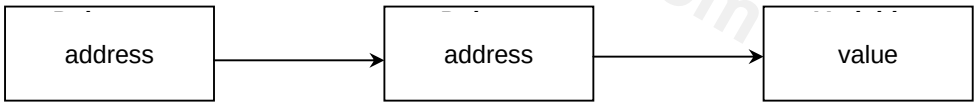
the function needs to know the number of columns so that it has much information about the end and start of rows within an array.

It is recommended to write programs to understand and practice the concepts of double dereferencing, single dereferencing, incrementing the name of the array to access different rows and columns etc. Only hands on practice will help understand the concept thoroughly.

Pointers to Pointers

What we have been talking about, now we will introduce a new terminology, is actually a case of 'Pointer to Pointer'. We were doing double dereferencing to access the elements of a two-dimensional array by using array name (a pointer) to access a row (another pointer) and further to access a column element (of 'int' data type).

In case of single dereference, the value of the pointer is the address of the variable that contains the value desired as shown in the following figure. In the case of pointer to pointer or double dereference, the first pointer contains the address of the second pointer, which contains the address of the variable, which contains the desired value.



Pointers to Pointers are very useful. But you need to be very careful while using the technique to avoid any problem.
Earlier, we used arrays and pointers interchangeably. We can think that a pointer to pointer is like a pointer to a group of arrays because a pointer itself can be considered as an array. We can elaborate with the following example by declaring character strings.
While using an array, we at first decide about the length of the array. For example, you are asked to calculate the average age of your class using the array. What would be the dimension of the array? Normally, you will look around, count the students of the class and keep the same size of the array as the number of students, say 53. Being a good programmer, you will look ahead and think about the maximum size of the class in the future and decide to take the size of the array as 100. Here, you have taken care of the future requirements and made the program flexible. But the best thing could be: to get the size of the array from the user at runtime and set it in the program

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

instead of declaring the array of maximum size. We will cover this topic at some later stage.

When we initialize an array with a character string, the number of characters in the character string determines the length of array (plus one character to include the '\0' character). eg. it is a single-dimensional array:

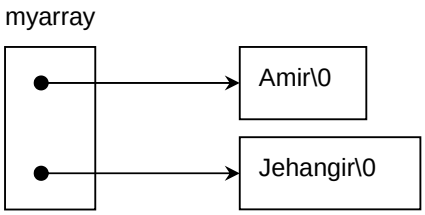
```
char name[] = "My full name";
```

The size of the 'name' array is 13.
Suppose, we have a group of character strings and we want to store them in a two-dimensional array. As we already discussed, an array has same number of columns in each row, e.g. a[5][10] array has 10 columns in each row. Now if we store character strings of variable length in a two-dimensional array, it is necessary to set the number of columns of the array as the length of the longest character string in the group (plus 1 byte for '\0' character). But the space within rows of the array would be wasted for all character strings with shorter length as compared to the number of columns. We don't want to waste this space and want to occupy the minimum space required to store a character string in the memory.
If we use the conventional two-dimensional array like a [5] [10], there is no way of using variable space for rows. All the rows will have fixed '10' number of columns in this case. But in case of an Array of Pointers, we can allocate variable space. An array of pointers is used to store pointers in it. Now we will try to understand how do we declare an array of pointers. The following statement can help us in comprehending it properly.

```
char * myarray[10];
```

We read it as: 'myarray is an array of 10 pointers to character'. If we take out the size of the array, it will become variable as:

```
char * myarray[] = {"Amir", "Jehangir"};
```



For first pointer myarray[0], 5 bytes (4 bytes for 'Amir' plus 1 byte for '\0') of memory has been allocated. For second pointer myarray[1], 9 bytes of memory is allocated. So this is variable allocation depending on the length of character string.

What this construct has done for us? If we use normal two-dimensional array, it will require fixed space for rows and columns. Therefore, we have used array of pointers here. We declared an array of pointers and initialized it with variable length character strings. The compiler allocates the same space as required for the character string to fit in. Therefore, no space goes waste. This approach has huge advantage.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

We will know more about Pointers to Pointers within next topic of Command-line Arguments and also in the case study given at the end of this lecture.

Command Line Arguments

Until now, we have always written the 'main()' function as under:

```
main( )
{
        . . . // code statements
}
```

But we are now in a position to write something inside the parenthesis of the 'main()' function. In C language, whenever a program is executed, the user can provide the command-line arguments to it like:

C:\Dev-cpp\work>Program-name argument1 argument2 ……argumentN

vubookshoppk.com

We have so far been taking input using the ‘cout’ and ‘cin’ in the program. But now we can also pass arguments from the command line just before executing the program. For this purpose, we will need a mechanism. In C, this can be done by using ‘argc’ and ‘argv’ arguments inside the main( ) function as:

```
void main(int argc, char **argv)
{
        . . .
}
```

Note that ‘argc’ and ‘argv’ are conventional names of the command line parameters of the ‘main()’ function. However, you can give the desired names to them.

argc = Number of command line arguments. Its type is ‘int’.
argv = It is a pointer to an array of character strings that contain the arguments, one per string. ‘**argv’ can be read as pointer to pointer to char.

vubookshoppk.com

This has been further explained in the following program. It counts down from a value specified on the command line and beeps when it reaches 0.

```
/* This program explains the use of command line  arguments */

#include <iostream.h>
#include <stdlib.h>     //Included for 'atoi( )' function

main(int argc, char **argv)
{
   int disp, count;
   if(argc < 2)
   {
      cout << "Enter the length of the count\n";
      cout << "on the command line. Try again.\n";
      return 1;
   }

   if(argc == 3 && !strcmp(*(argv + 2), "display"))
   {
      disp = 1;
   }
   else
   {
      disp = 0;
   }

   for(count = atoi(*(argv + 1)); count; --count)
   {
      if(disp)
      {
            cout << count <<' ';
      }
   }

   cout << '\a';  // '\a'causes the computer to beep

   return 0;
}
```

vubookshoppk.com

You must have noted that if no arguments are specified, an error message will be printed. It is common for a program that uses command-line arguments to issue instructions if an attempt has been made to run it without the availability of proper information. The first argument containing the number is converted into an integer using the standard function 'atoi( )'. Similarly, if the string 'display' is present as the second command-line argument, the count will also be displayed on the screen.

In theory, you can have up to 32,767 arguments but most operating systems do not allow more than a few because of the fixed maximum length of command-line. These

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

arguments are normally used to indicate a file name or an option. Using command-line arguments lends your program a very professional touch and facilitates the program's use in batch files.

Case Study: A Card Shuffling and Dealing Simulation

Now we want to move on to a real-world example where we can demonstrate pointer to pointer mechanism.

Problem:
Write a program to randomly shuffle the deck of cards and to deal it out.

Some Facts of Card Games:
- There are 4 suits in one deck: Hearts, Spades, Diamonds and Clubs.
- Each suit has 13 cards: Ace, Deuce, Three, Four, Five, Six, Seven, Eight, Nine, Ten, Jack, Queen and King.
- A deck has 13 * 4 = 52 cards in total.

Problem Analysis, Design and Implementation:
As obvious from the problem statement, we are dealing with the deck of cards, required to be identified. A card is identified by its suit i.e. it may be one of the Hearts, Spades, Diamonds or Clubs. Also every card has one value in the range starting from Ace to King. So we want to identify them in our program and our requirement is to use English like 'five of Clubs'. We will declare one array of suit like:

```
const char *suite[4] = {"Hearts", "Diamonds", "Clubs", "Spades" };
```

The second array is of values of cards:

```
const char *face[13] = { "Ace", "Deuce", "Three", "Four", "Five", "Six",
"Seven", "Eight", "Nine", "Ten", "Jack", "Queen" and "King"};
```

You must have noticed the use of array of pointers and 'const' keyword here. Both the arrays are declared in a way to avoid any wastage of space. Also notice the use of 'const' keyword. We declared arrays as constants because we want to use these values without modifying them.

Now we come to deck which has 52 cards. The deck is the one that is being shuffled and dealt. Definitely, it has some algorithmic requirements.
Firstly, what should be size and structure of the deck. It can either be linear array of 52 elements or 4 suites and 13 values (faces) per suit. Logically, it makes sense to have two-dimensional array of 4 suites and 13 faces per suit like:

```
int deck[4][13] = {0};
```

We will now think in terms of Algorithm Analysis.
The 'deck' is initialized with the 0 value, so that it holds no cards at start or it is empty. We want to distribute 52 cards. Who will load the 'deck' first, shuffle the cards and deal them out. How to do it?

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

As we want to select 52 cards (a deck) randomly, therefore, we can think of a loop to get one card randomly in every iteration. We will randomly choose one out of the 4 suites and select one value out of 13 values and store the card with its card number value in the deck. By this way, we will be writing numbers in the two-dimensional array of 'deck' randomly. That functionality is part of 'shuffle ()' function.

```
void shuffle( int wDeck[][13] )
{
        int row, column, card;

        for ( card = 1; card <= 52; card++){
                do{
                        row = rand() % 4;
                        column = rand() % 13;
                } while( wDeck [ row ][ column ] != 0 );
                wDeck[ row ][ column ] = card;
        }
}
```

You have noticed the 'rand()' function usage to generate random numbers. We are dividing the randomly generated number by 4 and 13 to ensure that we get numbers within our desired range. That is 0 to 3 for suites and 0 to 12 for values or faces. You also see the condition inside the 'while statement, 'wDeck[row][column] != 0 '. This is to ensure that we don't overwrite row and column, which has already been occupied by some card.

Now we want to deal the deck. How to deal it?
"At first, search for card number 1 inside the deck, wherever it is found inside the 'deck' array, note down the row of this element. Use this row to get the name of the suite from the 'suite' array. Similarly use the column to take out the value of the card from the 'face' array." See that the deal function is quite simple now.

```
void deal( const int wDeck[][ 13 ], const char *wFace[], const char *wSuit[])
{
        int card, row, column;
        for ( card = 1; card <= 52; card++ )
                for( row = 0; row <= 3; row++)
                        for( column = 0; column <= 12; column++)
                                if( wDeck[ row ][ column ] == card )
                                        cout << card << ".  " <<wFace[ column ]
                                << " of " << wSuit [row ] << '\n';

}
```

Here, we are not doing binary search that is more efficient. Instead, we are using simple brute force search. Also see the 'for loops' carefully and how we are printing the desired output.
Now we will discuss a little bit about the srand() function used while generating random numbers. We know that computers can generate random numbers through the 'rand()' function. Is it truly random? Be sure , it is not truly random. If you call

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

'rand()' function again and again. It will give you numbers in the same sequence. If you want your number to be really random number, it is better to set the sequence to start every time from a new value. We have used 'srand()' function for this purpose. It is a seed to the random number generator. Seed initializes the random number generator with a different value every time to generate true random numbers. We call 'srand()' function with a different value every time. The argument to 'srand()' function is taken from the 'time()' function which is giving us a new value after every one second. Every time we try to run the program, 'time()' returns a different number of seconds, which are passed to 'srand()' function as an argument so that the seed to the random number generator is a different number. It means that the random number generator now generates a different sequence of random numbers.

Although, you can copy this program and see the output after executing it, but this is not the objective of this exercise. You are required to study the problem and see the constructs very carefully. In this problem, you have examples of nested loops, array of pointers, variable sized strings in an array of pointers and random number usage in the real world problem etc.

```
/* Card shuffling and dealing program */

#include <iostream.h>
#include <stdlib.h>
#include <time.h>

void shuffle( int [] [ 13 ]);
void deal( const int [][ 13 ], const char *[], const char *[]);

int main()
{
    const char *suite[ 4 ] = {"Hearts", "Diamonds", "Clubs", "Spades" };
    const char *face[ 13 ] = { "Ace", "Deuce", "Three", "Four", "Five", "Six", "Seven",
"Eight", "Nine", "Ten", "Jack", "Queen", "King"};
    int deck[ 4 ][ 13 ] = { 0 };

    srand( time( 0 ) );

    shuffle( deck );
    deal( deck, face, suite );

    return 0;
}

void shuffle( int wDeck[][13] )
{
        int row, column, card;

        for ( card = 1; card <= 52; card++){
                do{
                        row = rand() % 4;
                        column = rand() % 13;
```

vubookshoppk.com

```
                    } while( wDeck [ row ][ column ] != 0 );

                wDeck[ row ][ column ] = card;
        }
}

void deal( const int wDeck[][ 13 ], const char *wFace[], const char *wSuit[])
{
     int card, row, column;
    const char *space;
    for ( card = 1; card <= 52; card++ )
        for( row = 0; row <= 3; row++)
                for( column = 0; column <= 12; column++)
                        if( wDeck[ row ][ column ] == card )
                          cout << card << ".   " <<wFace[ column ] << " of " << wSuit
[row ] << '\n';

}
```

A sample output of the program is:

```
1.  Six of Diamonds
2.  Ten of Hearts
3.  Nine of Clubs
4.  King of Hearts
5.  Queen of Clubs
6.  Five of Clubs
7.  Queen of Hearts
8.  Eight of Hearts
9.  Ace of Diamonds
10.  Ten of Diamonds
11.  Seven of Spades
12.  Ten of Clubs
13.  Seven of Clubs
14.  Three of Spades
15.  Deuce of Clubs
16.  Eight of Diamonds
17.  Eight of Clubs
18.  Nine of Spades
19.  Three of Clubs
20.  Jack of Clubs
21.  Queen of Spades
22.  Jack of Hearts
23.  Jack of Spades
24.  Jack of Diamonds
25.  King of Diamonds
26.  Seven of Hearts
27.  Five of Spades
28.  Seven of Diamonds
```

vubookshoppk.com

29. Deuce of Hearts
30. Ace of Spades
31. Five of Diamonds
32. Three of Hearts
33. Six of Clubs
34. Four of Hearts
35. Ten of Spades
36. Deuce of Spades
37. Three of Diamonds
38. Eight of Spades
39. Nine of Hearts
40. Ace of Clubs
41. Four of Spades
42. Queen of Diamonds
43. King of Clubs
44. Five of Hearts
45. Ace of Hearts
46. Deuce of Diamonds
47. Four of Diamonds
48. Four of Clubs
49. Six of Hearts
50. Six of Spades
51. King of Spades
52. Nine of Diamonds

vubookshoppk.com

Exercises

1. Write the program 'tail', which prints the last n lines of its input. By default, n is 10,
   let's say, but it can be changed by an optional argument, so that
   tail -n
   prints the last n lines.

Tips

- Pointers and arrays are closely related in C. The array names can be used as pointers but only as constant pointers.
- A function receiving a multi-dimensional array as a parameter must minimally define all dimensions except the leftmost one.
- Each time a pointer is incremented, it points to the memory location of the next element of its base type but in case of two-dimensional array, if you add some

number in a two-dimensional array name, it will jump as many rows as the added number.

- If the array is to be accessed in random order, then the pointer approach may not be better than array indexing.

- The use of pointers may reduce the wastage of memory space. As discussed in this lecture if we store a set of character strings of different lengths in a two-dimensional array, the memory space is wasted.

- Pointers may be arrayed (stored in an array) like any other data type.

- An array of pointers is the same as pointers to pointers.

- Although, you can give your desired names to the command line parameters inside 'main()' function but 'argc' and 'argv' are conventionally used.

vubookshoppk.com

Lecture No. 17

Reading Material

Deitel & Deitel - C++ How to Program

Chapter 5

5.29,5.30, 5.31, 5.32, 5.33, 5.34

Chapter 16

16.16 – 16.33 (Pages 869 – 884)

vubookshoppk.com

Summary

- String Handling
- String Manipulation Functions
  - Character Handling Functions
  - Sample Program
  - String Conversion Functions
  - String Functions
  - Search Functions
- Examples
- Exercises

String Handling

We have briefly talked about 'Strings' in some of the previous lectures. In this lecture, you will see how a string may be handled. Before actually discussing the subject, it is pertinent to know how the things were going on before the evolution of the concept of 'strings'.

When C language and UNIX operating system were being developed in BELL Laboratories, the scientists wanted to publish the articles. They needed a text editor to publish the articles. What they needed was some easy mechanism by which the articles could be formatted and published. We are talking about the times when PCs and word processors did not exist. It may be very strange thing for you people who can perform the tasks like making the characters bold, large or format a paragraph

with the help of word processors these days. Those scientists had not such a facility available with them. The task of writing article and turning into publishable material was mainly done with the help of typewriters. Then these computer experts decided to develop a program, which could help in the processing of text editing in an easy manner. The resultant efforts led to the development of a program for editing the text. The process to edit text was called text processing. The in- line commands were written as a part of the text and were processed on out put. Later, such programs were evolved in which a command was inserted for the functions like making the character bold. The effect of this command could be preview and then modified if needed.
Now coming to the topic of strings again, we will discuss in detail the in-built functions to handle the strings.

String Manipulation Functions

C language provides many functions to manipulate strings. To understand the functions, let's consider building block (or unit) of a string i.e., a character. Characters are represented inside the computers in terms of numbers. There is a code number for each character, used by a computer. Mostly the computers use ASCII (American Standard Code for Information Interchange) code for a character to store it. This is used in the computer memory for manipulation. It is used as an output in the form of character. We can write a program to see the ASCII values.
We have a data type *char* to store a character. A character includes every thing, which we can type with a keyboard for example white space, comma, full stop and colon etc all are characters. 0, 1, 2 are also characters. Though, as numbers, they are treated differently, yet they are typed as characters. Another data type is called as *int*, which stores whole numbers. As we know that characters are stored in side computer as numbers so these can be manipulated in the same form. A character is stored in the memory in one byte i.e. 8 bits. It means that 2^8 (256) different combinations for different values can be stored. We want to ascertain what number it stores, when we press a key on the board. In other words, we will see what character will be displayed when we have a number in memory.
The code of the program, which displays the characters and their corresponding integer, values (ASCII codes) as under.
In the program the statement c = *i* ; has integer value on right hand side (as i is an int) while **c** has its character representation. We display the value of **i** and **c**. It shows us the characters and their integer values.

```
//This program displays the ASCII code table

# include <iostream.h>

main ( )
{
        int i, char c ;
        for (i = 0 ; i < 256 ; i ++)
        {
                c = i ;
                cout << i << "\t" << c << "\n" ;
```

```
        }
}
```

In the output of this program, we will see integer numbers and their character representation. For example, there is a character, say white space (which we use between two words). It is a non-printable character and leaves a space. From the ASCII table, we can see that the values of a-z and A-Z are continuos. We can get the value of an alphabet letter by adding 1 to the value of its previous letter. So what we need to remember as a baseline is the value of 'a' and 'A'.

Character Handling Functions

C language provides many functions to perform useful tests and manipulations of character data. These functions are found in the header file **ctype.h**. The programs that have character manipulation or tests on character data must have included this header file to avoid a compiler error. Each function in **ctype.h** receives a character (an int) or EOF (end of file; it is a special character) as an argument. **ctype.h** has many functions, which have self-explanatory names.
Of these, **int isdigit (int c)** takes a simple character as its argument and returns true or false. This function is like a question being asked. The question can be described whether it is a character digit? The answer may be true or false. If the argument is a numeric character (digit), then this function will return true otherwise false. This is a useful function to test the input. To check for an alphabet (i.e. a-z), the function **isalpha** can be used. **isalpha** will return true for alphabet a-z for small and capital letters. Other than alphabets, it will return false. The function **isalnum** (is alphanumeric) returns true if its argument is a digit or letter. It will return false otherwise. All the functions included in **ctype.h** are shown in the following table with their description.

Prototype	Description
int isdigit(int c)	Returns **true** if **c** is a digit and **false** otherwise.
int isalpha(int c)	Returns **true** if **c** is a letter and **false** otherwise.
int isalnum(int c)	Returns **true** if **c** is a digit or a letter and **false** otherwise.
int isxdigit(int c)	Returns **true** if **c** is a hexadecimal digit character and **false** otherwise.
int islower(int c)	Returns **true** if **c** is a lowercase letter and **false** otherwise.
int isupper(int c)	Returns **true** if **c** is an uppercase letter; **false** otherwise.
int tolower(int c)	If **c** is an uppercase letter, **tolower** returns **c** as a lowercase letter. Otherwise, **tolower** returns the argument unchanged.
int toupper(int c)	If **c** is a lowercase letter, **toupper** returns **c** as an uppercase letter. Otherwise, **toupper** returns the argument unchanged.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

int isspace(int c)	Returns **true** if **c** is a white-space character—newline (**'\n'**), space (**' '**), form feed (**'\f'**), carriage return (**'\r'**), horizontal tab (**'\t'**), or vertical tab (**'\v'**)—and **false** otherwise
int iscntrl(int c)	Returns **true** if **c** is a control character and **false** otherwise.
int ispunct(int c)	Returns **true** if **c** is a printing character other than a space, a digit, or a letter and **false** otherwise.
int isprint(int c)	Returns **true** value if **c** is a printing character including space (**' '**) and **false** otherwise.
int isgraph(int c)	Returns **true** if **c** is a printing character other than space (**' '**) and **false** otherwise.

The functions **tolower** and **toupper** are conversion functions. The **tolower** function converts its uppercase letter argument into a lowercase letter. If its argument is other than uppercase letter, it returns the argument unchanged. Similarly the **toupper** function converts its lowercase letter argument into uppercase letter. If its argument is other than lowercase letter, it returns the argument without effecting any change.

Sample Program

Let's consider the following example to further demonstrate the use of the functions of **ctype.h**. Suppose, we write a program which prompts the user to enter a string. Then the string entered is checked to count different types of characters (digit, upper and lowercase letters, white space etc). We keep a counter for each category of character entered. When the user ends the input, the number of characters entered in different types will be displayed. In this example we are using a function **getchar()**, instead of **cin** to get the input. This function is defined in header file as **stdio.h**. While carrying out character manipulation, we use the **getchar()** function. This function reads a single character from the input buffer or keyboard. This function can get the new line character '\n' (the ENTER key) so we run the loop for input until user presses the ENTER key. As soon as the **getchar()** gets the ENTER key pressed (i.e. new line character '\n'), the loop is terminated. We know that, every C statement returns a value. When we use an assignment statement (as used in our program **c = getchar())**, the value assigned to the left hand side variable is the value of the statement too. Thus, the statement **(c = getchar())** returns the value that is assigned to char **c**. Afterwards, this value is compared with the new line character '\n'. If it is not equal inside the loop, we apply the tests on **c** to check whether it is uppercase letter, lowercase letter or a digit etc. In this program, the whole string entered by the user is manipulated character.

Following is the code of this program.

```
// Example: analysis of text using <ctype.h> library

#include <iostream.h>
#include <stdio.h>
#include <ctype.h>
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
main()
{
        char c;
        int i = 0, lc = 0, uc = 0, dig = 0, ws = 0, pun = 0, oth = 0;

        cout << "Please enter a character string and then press ENTER:  ";

        // Analyse text as it is input:

        while ((c = getchar()) != '\n')
        {
                if (islower(c))
                        lc++;
                else if (isupper(c))
                        uc++;
                else if (isdigit(c))
                        dig++;
                else if (isspace(c))
    ws++;
        else if (ispunct(c))
    pun++;
                else
                        oth++;
        }
        // display the counts of different types of characters
cout << "You typed:"<< endl;
     cout<< "lower case letters =  "<<  lc<< endl;
     cout << "upper case letters =  " << uc <<endl;
cout<< "digits =  " << dig << endl;
cout<< "white space =  "<< ws << endl;
cout<< "punctuation =  "<< pun<< endl;
cout<< "others =  "<< oth;
}
```

A sample output of the program is given below.

```
Please enter a character string and then press ENTER:  Sixty Five = 65.00
You typed:
lower case letters =  7
upper case letters =  2
digits =  4
white space =  3
punctuation =  2
others =  0
```

String Conversion Functions

vubookshoppk.com

The header file **stdlib.h** includes functions, used for different conversions. When we get input of a different type other than the type of variable in which the value is being stored, it warrants the need to convert that type into another type. These conversion functions take an argument of a type and return it after converting into another type. These functions and their description are given in the table below.

Prototype	Description
double atof(const char *nPtr)	Converts the string **nPtr** to **double**.
Int atoi(const char *nPtr)	Converts the string **nPtr** to **int**.
long atol(const char *nPtr)	Converts the string **nPtr** to long **int**.
double strtod(const char *nPtr, char **endPtr)	Converts the string **nPtr** to **double**.
long strtol(const char *nPtr, char **endPtr, int base)	Converts the string **nPtr** to **long**.
unsigned long strtoul(const char *nPtr, char **endPtr, int base)	Converts the string **nPtr** to **unsigned long**.

Use of these functions:
While writing main () in a program, we can put them inside the parentheses of main. 'int arg c, char ** arg v are written inside the parentheses. The **arg c** is the count of number of arguments passed to the program including the name of the program itself while **arg v** is a vector of strings or an array of strings. It is used while giving command line arguments to the program. The arguments in the command line will always be character strings. The number in the command line (for example 12.8 or 45) are stored as strings. While using the numbers in the program, we need these conversion functions.

Following is a simple program which demonstrate the use of **atoi** function. This program prompts the user to enter an integer between 10-100, and checks if a valid integer is entered.

```
//This program demonstrate the use of atoi function

# include <iostream.h>
# include <stdlib.h>

main( )
{
        int anInteger;
        char myInt [20]
        cout << "Enter an integer between 10-100 : ";
        cin >> myInt;
        if (atoi(myInt) == 0)
                cout << "\nError : Not a valid input"; // could be non numeric
        else
        {
                anInteger = atoi(myInt);
                if (anInteger < 10 || anInteger > 100)
                        cout << "\nError : only integers between 10-100 are allowed!";
                else
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
                cout << "\n OK, you have entered " << anInteger;
        }
}
```

The output of the program is as follows.

```
Enter an integer between 10-100 : 45.5
OK, you have entered 45
```

String Functions

We know a program to guess a number, stored in the computer. To find out a name (which is a character array) among many names in the memory, we can perform string comparison on two strings by comparing a character of first string with the corresponding character of the second string. Before doing this, we check the length of both the strings to compare. C library provides functions to compare strings, copy a string and for other string manipulations.
The following table shows the string manipulation functions and their description. All these functions are defined in the header file **string.h**, in the C library.

Function prototype	Function description
char *strcpy(char *s1, const char *s2)	Copies string **s2** into character array **s1**. The value of **s1** is returned.
char *strncpy(char *s1, const char *s2, size_t n)	Copies at most **n** characters of string **s2** into array **s1**. The value of **s1** is returned.
char *strcat(char *s1, const char *s2)	Appends string **s2** to array **s1**. The first character of **s2** overwrites the terminating null character of **s1**. The value of **s1** is returned.
char *strncat(char *s1, const char *s2, size_t n)	Appends at most **n** characters of string **s2** to array **s1**. The first character of **s2** overwrites the terminating null character of **s1**. The value of **s1** is returned.
int strcmp(const char *s1, const char *s2)	Compares string **s1** to **s2.** Returns a negative number if **s1 < s2**, zero if **s1 == s2** or a positive number if **s1 > s2**
int strncmp(const char *s1, const char *s2, size_t n)	Compares up to **n** characters of string **s1** to **s2**. Returns a negative number if **s1 < s2**, zero if **s1 == s2** or a positive number if **s1 > s2.**
int strlen (const char *s)	Determines the length of string s. The number of characters preceding the terminating null character is returned.

Let's look at the string copy function which is **strcpy**. The prototype of this function is

char *strcpy(char *s1, const char *s2)

Here the first argument is a pointer to a character array or string s1 whereas the second argument is a pointer to a string s2. The string s2 is copied to string s1 and a pointer to that resultant string is returned. The string s2 remains the same. We can describe the string s1 as the destination string and s2 as the source string. As the source remains the same during the execution of **strcpy** and other string functions, the **const** keyword is used before the name of source string. The **const** keyword prevents any change in the source string (i.e. s2). If we want to copy a number of characters of a string instead of the entire string, the function **strncpy is employed.** The function **strncpy** has arguments a pointer to destination strings (s1), a pointer to source string (s2) . The third argument is **int n**. Here **n** is the number of characters which we want to copy from **s2** into **s1**. Here **s1** must be large enough to copy the **n** number of characters.

The next function is **strcat** (string concatenation). This function concatenates (joins) two strings. For example, in a string, we have first name of a student, followed by another string, the last name of the student is found. We can concatenate these two strings to get a string, which holds the first and the last name of the student. For this purpose, we use the **strcat** function. The prototype of this function is **char *strcat(char *s1, const char *s2)**. This function writes the string s2 (source) at the end of the string s1(destination). The characters of s1 are not overwritten. We can concatenate a number of characters of s2 to s1 by using the function **strncat**. Here we provide the function three arguments, a character pointer to s1, a character pointer to s2 while third argument is the number of characters to be concatenated. The prototype of this function is written as

char *strncat(char *s1, const char *s2, size_t n)

Examples

Let's consider some simple examples to demonstrate the use of **strcpy, strncpy, strcat and strncat** functions. To begin with, we can fully understand the use of the function **strcpy** and **strncpy**.

Example 1

```
//Program to display the operation of the strcpy() and strncpy()

# include<iostream.h>
# include<string.h>

void main()
{
char string1[15]="String1";
char string2[15]="String2";

cout<<"Before the copy :"<<endl;
cout<<"String 1:\t"<<string1<<endl;
cout<<"String 2:\t"<<string2<<endl;

        //copy the whole string
strcpy(string2,string1);          //copy string1 into string2
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
cout<<"After the copy :"<<endl;
cout<<"String 1:\t"<<string1<<endl;
cout<<"String 2:\t"<<string2<<endl;

        //copy three characters of the string1 into string3
        strncpy(string3, string1, 3);
        cout << "strncpy (string3, string1, 3)  =   " << string3 ;
}
```

Following is the output of the program.

```
Before the copy :
String 1:       String1
String 2:       String2
After the copy :
String 1:       String1
String 2:       String1
Strncpy (string3, string1, 3)  =   Str
```

Example 2 (strcat and strncat)

The following example demonstrates the use of function **strcat** and **strncat.**

```
//Program to display the operation of the strcat() and strncat()

#include <iostream.h>
#include <string.h>

int main()
{
char s1[ 20 ] = "Welcome to ";
char s2[] = "Virtual University ";
char s3[ 40 ] = "";
cout<< "s1 =  " << s1 << endl << "s2 =  " << s2 << endl << "s3 = " << s3 << endl;
cout<< "strcat( s1, s2 ) =   "<< strcat( s1, s2 );
cout << "strncat( s3, s1, 6 ) =   " << strncat( s3, s1, 6 );
}
```

The output of the program is given below.

```
s1 =  Welcome to
s2 =  Virtual University
s3 =
strcat( s1, s2 ) =   Welcome to Virtual University
strncat( s3, s1, 7 ) =   Welcome
```

vubookshoppk.com

Now we come across the function **strcmp.** This function compares two strings, and returns an integer value depending upon the result of the comparison. The prototype of this function is **int strcmp(const char *s1, const char *s2)**
This function returns a number less than zero (a negative number), if s1 is less than s2. It returns zero if s1 and s2 are identical and returns a positive number (greater than zero) if s1 is greater than s2. The space character in a string and lower and upper case letters are also considered while comparing two strings. So the strings "Hello", "hello" and "He llo" are three different strings these are not identical.
Similarly there is a function **strncmp**, which can be used to compare a number of characters of two strings. The prototype of this function is

int strncmp(const char *s1, const char *s2, size_t n)

Here s1 and s2 are two strings and **n** is the number upto which the characters of s1 and s2 are compared. Its return type is also **int**. It returns a negative number if first n characters of s1 are less than first n characters of s2. It returns zero if n characters of s1 and n characters of s2 are identical. However, it returns a positive number if n characters of s1 are greater than n characters of s2.
Now we will talk about the function, '**strlen'** (string length) which is used to determine the length of a character string. This function returns the length of the string passed to it. The prototype of this function is given below.

int strlen (const char *s)

This function determines the length of string s. the number of characters preceding the terminating null character is returned.

Search Functions

C provides another set of functions relating to strings, called search functions. With the help of these functions, we can do different types of search in a string. For example, we can find at what position a specific character exists. We can search a character starting from any position in the string. We can find the preceding or proceeding string from a specific position. We can find a string inside another string. These functions are given in the following table.

Function prototype	Function description
char *strchr(const char *s, int c);	Locates the first occurrence of character **c** in string **s**. If **c** is found, a pointer to **c** in **s** is returned. Otherwise, a **NULL** pointer is returned.
size_t strcspn(const char *s1, const char *s2);	Determines and returns the length of the initial segment of string **s1** consisting of characters not contained in string **s2**.
size_t strspn(const char *s1, const char *s2);	Determines and returns the length of the initial segment of string **s1** consisting only of characters contained in string **s2**.
char *strpbrk(const char *s1, const char *s2);	Locates the first occurrence in string **s1** of any character in string **s2**. If a character from string **s2** is found, a pointer to the character in string **s1** is returned. Otherwise, a **NULL** pointer is returned.
char *strrchr(const char *s, int c);	Locates the last occurrence of **c** in string **s**. If **c** is found, a pointer to **c** in string **s** is returned.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

	Otherwise, a **NULL** pointer is returned.
char *strstr(const char *s1, const char *s2);	Locates the first occurrence in string **s1** of string **s2**. If the string is found, a pointer to the string in **s1** is returned. Otherwise, a **NULL** pointer is returned.
char *strtok(char *s1, const char *s2);	A sequence of calls to **strtok** breaks string **s1** into "tokens"—logical pieces such as words in a line of text—separated by characters contained in string **s2**. The first call contains **s1** as the first argument, and subsequent calls to continue tokenizing the same string contain **NULL** as the first argument. A pointer to the current token is returned by each call. If there are no more tokens when the function is called, **NULL** is returned.

Example 3

Here is an example, which shows the use of different string manipulation functions. The code of the program is given below.

```
//A program which shows string manipulation using <string.h> library

#include <iostream.h>
#include <string.h>
#include <stdlib.h>

main()
{
char s1[] = "Welcome to " ;
char s2[] = "Virtual University" ;
char s3[] = "Welcome to Karachi" ;
char city[] = "Karachi";
char province[] = "Sind";
      char s[80];
char *pc;
          int n;

          cout << "s1 = " << s1 << endl << "s2 = " << s2 << endl ;
          cout << "s3 = " << s3 << endl ;
          // function for string length
          cout << "The length of s1 = " << strlen(s1) << endl ;
          cout << "The length of s2 = " << strlen(s2) << endl ;
          cout << "The length of s3 = " << strlen(s3) << endl ;

          strcpy(s, "Hyderabad");  // string copy
          cout<< "The nearest city to "<< city << " is " << s << endl ;

          strcat(s, " and ");  // string concatenation
          strcat(s,city);
          strcat(s, " are in ");
          strcat(s, province);
          strcat(s, ".\n");
```

vubookshoppk.com

```
        cout << s;

        if (!(strcmp (s1,s2)))   // ! is used as zero is returned if s1 & s2 are equal
                cout << "s1 and s2 are identical" << endl ;
        else
                cout << "s1 and s2 are not identical" << endl ;

        if (!(strncmp (s1,s3,7)))        // ! is used as zero is returned for equality
                cout << "First 7 characters of s1 and s3 are identical" << endl ;
        else
                cout << "First 7 characters of s1 and s3 are not identical" << endl ;
}
```

Following is the output of the program.

```
S1 = Welcome to
S2 = Virtual University
S3 = Welcome to Karachi
The length of s1 = 11
The length of s2 = 18
The length of s3 = 18
The nearest city to Karachi is Hyderabad
Hyderabad and Karachi are in Sind.
S1 and s2 are not identical
First 7 characters of s1 and s3 are identical
```

vubookshoppk.com

Exercises

1: Write a program that displays the ASCII code set in tabular form on the screen.
2: Write your own functions for different manipulations of strings.
3: Write a program, which uses different search functions.

Lecture No. 18

Reading Material

Deitel & Deitel - C++ How to Program Chapter 14
14.3, 14.4, 14.5, 14.6

Summary

Files
Text File Handling
Example 1
Output File Handling
Example 2
Tips

Files

Today's topic is about files and file handling. We have been talking about bit, bytes, character, numbers etc. Then we discuss about strings, which are actually character arrays. When we combine all these things, it becomes a program. We type a letter or a document in word processor. Similarly, we can have a novel as a combinations of characters, words, sentences, bigger collection of words and sentences. These are no longer bits and bytes. We call these sentences, paragraphs as files. There are many types of files in computer. Primarily, there are two types of files i.e. text files and executable program files. Text files consist of readable English characters. These include our simple text files, or word processor file etc. On the other hand, the executable program files run the program. In the dos (command prompt window), when we issue the command 'dir', a list of files is displayed. Similarly, Windows explorer is used in the windows, click on some folder to see the list of the files in that folder in the right panel. These are the names of the files, which we see. The file properties show the length of the file, date of creation etc. One category of data files is plain text files. We can create plain text files using the windows note pad, type the text and save it. It is an ordinary text, meaning that there is no formatting of text involved and we can view this text using the 'type' command of the dos (type filename). Similarly, our source programs are also plain text files. There is no formatted text in cpp files. There is another kind of text files, which are not plain ones. These are the word processor files, containing text that is formatted like, bold, italic, underline, colored text and tables. This formatting information is also stored in the file along with the text. Therefore such files are not plain text files. Same thing applies to spreadsheets having formatting, formulae, cell characteristic etc. Though these files contain some binary information along with the text, yet these are not program files. We created these files using some other program like Microsoft Word, excel etc. Such files also fall in the category of text files. The other type is the program file that executes on the computer. Normally, executable files contain only non-printable binary information. There are different ways of handling these files. Today we will see what is the utility of files in our programs. We know that all the information in the computer memory is volatile. It means when we turn off the computer that information will be lost. The data, written in a program, is actually the part of the program and is saved on the disk. Whenever we execute the program that

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

data will be available. Suppose we have to develop a payroll system for a factory. For this purpose, we will at first need to gather the data like name of the employees, their salaries etc. Enter all this information before getting their attendance. After collecting all the information, you can calculate their salary and print a report of the salary. Now the question arises whether we will have to enter the name and salary of employees every month. The better way is to store this information once and re-use it every month. We can save this information in a file and can calculate the salary after getting the current month's attendance of employees. We have to do all the calculations again in case of not saving the report on the disk. It will be nicer if we have saved the output file on the disk. We can take the print out whenever we need. We are discussing this just to give you the justification of using files. The data in the memory is volatile. Similarly, the data, which we key in the program during the execution of a program, is also volatile. To save the data on permanent basis, we need files so that we keep these on the disk and can use whenever needed. Now there is need to learn how to create a file on the disk, read from the file, and write into the file and how to manipulate the data in it. This is the file handling.

Text file Handling

Let's look what are the basic steps we need for file handling. Suppose we have a file on the disk and want to open it. Then read from or write into the file before finally closing it. The basic steps of file handling are:

Open the file
Read and write
Close the file

We have been using *cin* and *cout* a lot in our programs. We know that these are the doors by which data can enter and come out. *cin* is used to enter the data and *cout* is used to display the data on the screen. Technically, these are known as streams in C++. We will discuss in detail about streams in later lectures. Today we will see some more streams about file handling. This is how *'C++ language'* handles files. For this purpose, the header file to be used is <*fstream.h*> (i.e. file stream). Whenever using files in our program, we will include this header file as *#include* <*fstream.h*>. These streams are used the way we have been employing *cin* and *cout* but we can do more with these streams. While handling files, one can have three options. Firstly, we will only read the file i.e. read only file. It means the file is used as input for the program. We need to have a stream for input file i.e. *ifstream* (input file stream). Similarly, if we want to write in some file, *ofstream* (output file stream) can be used. Sometimes we may need to read and write in the same file. One way is to read from a file, manipulate it and write it in another file, delete the original file and renaming the new file with the deleted file name. We can read, write and manipulate the same file using *fstream.h*.

Let us take a look how can we use these files in our programs. First, we have to include the *fstream.h* in our programs. Then we need to declare file streams. *cin* and *cout* are predefined streams therefore we did not declare these. We can declare file stream as:

```
ifstream inFile;        // object for reading from a file
ofstream outFile;       // object for writing to a file
```

vubookshoppk.com

The variables *inFile* and *outFile* are used as handle to refer files. These are like internal variables which will be used to handle the files that are on the disk. If we want to read a file, we will use *inFile* as declared above to read a file. Any meaningful and self-explanatory name can be used. To deal with a payroll system, *payrollDataFile* can be used as a file stream variable i.e. *ifstream payrollDataFile;*. Consider the following statement:

```
ifstream   myFile;
```

Here *myFile* is an internal variable used to handle the file. So far, we did not attach a file with this handle. Before going for attachment, we will have to open a file. Logically, there is function named 'open' to open a file. While associating a file with the variable *myFile,* the syntax will be as under:

```
myFile.open(filename);
```

You have noted that this is a new way of function calling. We are using dot (.) between the *myFile* and *open* function. *myFile* is an object of *ifstream* and *open()* is a function of *ifstream*. The argument for the *open* function *filename* is the name of the file on the disk. The data type of argument *filename* is character string, used to give the file name in double quotation marks. The file name can be simple file name like "payroll.txt". It can be fully qualified path name like "C:\myprogs\payroll.txt". In the modern operating systems like Windows, disks are denoted as C: or D: etc. We have different folders in it like 'myprogs' and can have files in this folder. The fully qualified path means that we have to give the path beginning from C:\.
To under stand it further, suppose that we are working in the folder 'myprogs' and our source and executable files are also in this folder. Here, we don't need to give a complete path and can write it as "*payroll.txt*". If the file to be opened is in the current directory (i.e. the program and text file are in the same folder), you can open it by simply giving the name. If you are not familiar with the windows file system, get some information from windows help system. It is a hierarchical system. The disk, which is at the top, contains folder and files. Folders can contain subfolders and files. It is a multi-level hierarchical system. In UNIX, the top level is "root", which contains files and directories. So it's like a bottom-up tree. Root is at the top while the branches are spreading downward. Here 'root' is considered as root of a tree and files or subfolders are branches.

To open a file, we use *open* function while giving it the name of the file as fully qualified path name or simple name. Then we also tell it what we want to do with that file i.e. we want to read that file or write into that file or want to modify that file. We have declared *myFile* as *ifstream* (input file stream) variable so whenever we tried to open a file with *ifstream* variable it can only be opened for input. Once the file is open, we can read it. The access mechanism is same, as we have been using with streams. So to read a word from the file we can write as:

```
myFile >> c;
```

So the first word of the file will be read in c, where c is a character array. It is similar as we used with *cin*. There are certain limitations to this. It can read just one word at one time. It means, on encountering a space, it will stop reading further. Therefore,

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

we have to use it repeatedly to read the complete file. We can also read multiple words at a time as:

```
myFile >> c1 >> c2 >> c3;
```

The first word will be read in c1, 2nd in c2 and 3rd in c3. Before reading the file, we should know some information regarding the structure of the file. If we have a file of an employee, we should know that the first word is employee's name, 2nd word is salary etc, so that we can read the first word in a *string* and 2nd word in an *int* variable. Once we have read the file, it must be closed. It is the responsibility of the programmer to close the file. We can close the file as:

```
myFile.close();
```

The function *close()* does not require any argument, as we are going to close the file associated with *myFile.* Once we close the file, no file is associated with *myfile* now.

Lets take a look on error checking mechanism while handling files. Error checking is very important. Suppose we have to open a text file *myfile.txt* from the current directory, we will write as:

```
ifstream myFile;
myFile.open("myfile.txt");
```

If this file does not exist on the disk, the variable *myFile* will not be associated with any file. There may be many reasons due to which the *myFile* will not be able to get the handle of the file. Therefore, before going ahead, we have to make sure that the file opening process is successful. We can write as:

```
if (!myFile)
{
        cout << "There is some error opening file" << endl;
        cout << " File cannot be opened" << end;
        exit(1);
}
else
        cout << " File opened successfully " << end;
```

Example 1

Let's write a simple program, which will read from a file *'myfile.txt'* and print it on the screen. *"myfile.txt"* contains employee's name, salary and department of employees. Following is the complete program along with "myfile.txt" file.

Sample "myfile.txt".

```
Name  Salary  Department
Aamir  12000  Sales
Amara 15000  HR
Adnan 13000  IT
Afzal   11500  Marketing
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Code of the program.

```
/*
* This program reads from a txt file "myfile.txt" which contains the
* employee information
*/

#include <iostream.h>
#include <fstream.h>

main()
{
   char name[50];       // used to read name of employee from file
   char sal[10];        // used to read salary of employee from file
   char dept[30];       // used to read dept of employee from file
   ifstream inFile;        // Handle for the input file

   char inputFileName[] = "myfile.txt";     // file name, this file is in the current directory

   inFile.open(inputFileName);              // Opening the file

   // checking that file is successfully opened or not
   if (!inFile)
   {
          cout << "Can't open input file named " << inputFileName << endl;
          exit(1);
   }

   // Reading the complete file word by word and printing on screen
  while (!inFile.eof())
   {
           inFile >> name >> sal >> dept;
           cout << name << "\t" << sal << " \t" << dept << endl;
   }
   inFile.close();
}
```

Output of the program.

```
Name    Salary   Department
Aamir    12000   Sales
Amara   15000   HR
Adnan   13000   IT
Afzal    11500   Marketing
```

In the above program, we have declared three variables for reading the data from the input file (i.e. name, sal, dept). The text file "myfile.txt" and the program file should be in the same directory as there is no fully qualified path used with the file name in the *open()* function. After opening the file, we will check that file is successfully opened or not. If there is some error while opening the file, we will display the error on screen and exit from the program. The statement *exit(1)* is used to exit from the

vubookshoppk.com

program at any time and the control is given back to the operating system. Later, we will read all the data from the file and put it into the variables. The condition in 'while loop' is *"!inFile.eof()"* means until the end of file reached. The function *eof()* returns true when we reached at the end of file.

Output File Handling

Let's talk about the output file handling. You can do several things with output files like, creation of a new file on the disk and writing data in it. Secondly, we may like to open an existing file and overwrite it in such a manner that all the old information is lost from it and new information is stored. Thirdly, we may want to open an existing file and append it in the end. Fourthly, an existing file can be opened and modified in a way that it can be written anywhere in the file. Therefore, when we open a file for output we have several options and we might use any one of these methods. All these things are related to the file-opening mode. The actual syntax of open function is:

```
open (filename, mode)
```

The first argument is the name of the file while the second will be the mode in which file is to be opened. Mode is basically an integer variable but its values are pre-defined. When we open a file for input, its mode is input file that is defined and available through the header files, we have included. So the correct syntax of file opening for input is:

```
myFile.open("myfile.txt" , ios::in);
```

The 2nd argument *ios::in* associates *myFile* stream object with the "*myfile.txt*" for input. Similarly, for output files, there are different modes available. To open a file for output mode, *ios::out* is used. Here is the complete list of modes:

vubookshoppk.com

Mode	Meaning
in	Open a file or stream for extraction (input)
out	Open a file or stream for insertion (output)
app	Append rather than truncate an existing file. Each insertion (output) will be written to the end of the file
trunc	Discards the file's contents if it exists. (similar to default behavior)
ate	Opens the file without truncating, but allows data to be written anywhere in the file
binary	Treat the file as binary rather than text. A binary file has data stored in internal formats, rather than readable text format

If a file is opened with *ios::out* mode, a new file is created. However, if the file already exists, its contents will be deleted and get empty unless you write something into it. If we want to append into the file, the mode will be *ios::app.* When we write into the file, it will be added in the end of the file. If we want to write anywhere in the file, the mode is *ios::ate*. We can position at some particular point and can write there.

It is like append mode. But in '*ate* mode' we can write anywhere in the file. With the *trunc* mode, the file is truncated, it is similar to *out* mode.

Exercise:
Write a program, which creates a new file, and write "Welcome to VU" in it.

The code of the program is:

```
/*
* This program writes into a txt file "myfileOut.txt" which contains the
* "Welcome to VU"
*/

#include <iostream.h>
#include <fstream.h>

main()
{
   ofstream outFile;                              // Handle for the input file
   char outputFileName[] = "myFileOut.txt";  // The file is created in the current directory
   char ouputText[100] = "Welcome to VU";  // used to write into the file

   outFile.open(outputFileName, ios::out);      // Opening the file

   // checking that file is successfully opened or not
   if (!outFile)
   {
            cout << "Can't open input file named " << outputFileName << endl;
            exit(1);
   }

   // Writing into the file
   outFile << ouputText;
   outFile.close();
}
```

The file "myFileOut.txt":

```
Welcome to VU
```

Exercise:
Write a program, which reads an input file of employee's i.e. "employeein.txt", add the salary of each employee by 2000, and write the result in a new file "employeeout.txt".

The sample input file "employeein.txt"

```
Aamir  12000
Amara 15000
Adnan 13000
Afzal   11500
```

vubookshoppk.com

The output file "employeeout.txt" should be as:

```
Name  Salary
Aamir 14000
Amara 17000
Adnan 15000
Afzal  13500
```

We have been using '>>' sign for reading data from the file. There are some other ways to read from the file. The *get()* function is used to get a character from the file, so that we can use *get()* to read a character and put it in a char variable. The last character in the file is *EOF*, defined in header files. When we are reading file using *get()* function the loop will be as:

```
char c;
while ( (c = inFile.get()) != EOF)
{
        // do all the processing
        outFile.put(c);
}
```

There is one limitation with the '>>' and that is it does not read the new line character and in the output file we have to insert the new line character explicitly, whereas *get()* function reads each character as it was typed. So if we have to make a copy of a file, the function *get()* should be used. Can we have a function to put a character in the output file? Yes, the function to write a single character in the out put file is *put()*, so with the output file stream handle, we can use this function to write a character in the output file.

Exercise:
Write the above programs using the *get()* function and verify the difference of '>>' and 'get()' using different input files.

Whenever we declare a variable we initialize it like if we declare an integer as *int i.* We initialize it as *i = 0.* Similarly we can declare and initialize an input or output file stream variable as:

```
ifstream inFile("myFileIn.txt");
ofstream outFile("myfileOut.txt", ios::out);
```

This is a short hand for initialization. This is same as we open it with *open()* function. Normally we open a file explicitly with the *open()* function and close it explicitly with *close()* function. Another advantage of using explicitly opening a file using the *open()* function is, we can use the same variable to associate with other files after closing the first file.

We can also read a line from the file. The benefit of reading a line is efficiency, but clarity should not be sacrificed over efficiency. We read from the disk and write to the disk. The disk is an electro mechanical device and is the slowest component in the computer. Other parts like processors, memory etc are very fast nowadays i.e. up o 2Ghz. When we talk about hard disk, we say its average access time is 7 mili sec. It

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

means when we request hard disk to get data it will take 7 mili sec (7/1000 of a sec) to get the data where as processor is running on GHz speed which is thousand million cycles per sec. Processor and memory are much much faster than the hard disk. Therefore reading a single character from the file is too slow. Although nowadays, the buffering and other techniques are used to make the disk access faster. It will be quite efficient if we read the data in bigger chunks i.e. 64k or 256k bytes and also write in bigger chunks. Today's operating system applies the buffering and similar techniques. Instead of reading and writing character-by-character or word-by-word, reading and writing line by line is efficient. A function is available for this purpose i.e. *getLine()* for input file stream and *putLine()* for output file stream. The syntax of *getLine()* is as follows:

```
char name[100];
int maxChar = 100;
int stopChar = 'o';
inFile.getLine(name, maxChar, stopChar);
```

The first argument is a character array, the array should be large enough to hold the complete line. The second argument is the maximum number of characters to be read. The third one is the character if we want to stop somewhere. Suppose we have an input file containing the line 'Hello World', then the statements:

```
char str[20];
inFile.getLine(str, 20, 'W');
cout << "The line read from the input file till W is " << str;
```

The *getLine()* function will read 'Hello '. Normally we do not use the third argument. The default value for the third argument is new line character so *getLine()* will read the complete line up to the new line character. The new line character will not be read. The line read will be stored in the array, used in the first argument. It is our responsibility that the array should be large enough to hold the entire line and then we can manipulate this data. Using the *getLine()* repeatedly to read the file is much more efficient rather than using the *get()* function. As the *getLine()* function does not read the new line character, we have to put it explicitly. If we have large file to be read then difference in speed with both the programs i.e. using *get()* and *getLine()* can be noted.

Exercise:
Write a program which reads a file using the getLine() function and display it on the screen.

Sample input file:

```
This is a test program
In this program we learn how to use getLine() function
This function is faster than using the get() function
```

The complete code of the program:

```
/*
* This program reads from a txt file line by line
*
```

vubookshoppk.com

```
*/

#include <iostream.h>
#include <fstream.h>

main()
{
   ifstream inFile;                          // Handle for the input file
   char inputFileName[] = "test.txt";        // file name, this file is in the current directory
   const int MAX_CHAR_TO_READ = 100;    // maximum character to read in one line
   char completeLineText[MAX_CHAR_TO_READ];  // to be used in getLine function

   inFile.open(inputFileName);               // Opening the file

   // checking that file is successfuly opened or not
   if (!inFile)
   {
          cout << "Can't open input file named " << inputFileName << endl;
          exit(1);
   }

   // Reading the complete file line by line and printing on screen

   while (!inFile.eof())
   {
        inFile.getline(completeLineText, MAX_CHAR_TO_READ);
        cout << completeLineText << endl;
   }
   inFile.close();
}
```

vubookshoppk.com

The output of the program is:

```
This is a test program
In this program we learn how to use getLine() function
This function is faster than using the get() function
```

Example 2

Problem statement:

A given input file contains Name of the employee and salary of current month. There is a single space between the name and the salary. Name and salary can not contain spaces. Calculate the total salaries of the employees. Create an output file and write the total salary in that file.

Solution:

We can read a line from the input file using the *getLine()* function. Now we need to break this line into pieces and get the name and salary in different variables. Here we can use the string token function i.e. *strtok().* The string token function (strtok()) takes a string and a delimiter i.e. the character that separates tokens from each other. As there is a space between the name and the salary, we can use the space character as delimiter. So the first call to the string token function will return the name of the

employee, the second call will return the salary of the employee. The syntax to get the next token from the *strtok()* function is: strtok(NULL, ' '), it means return the next token from the same string. The second token contains the salary of the employee and is in a char string. We need to add the salaries of all the employees. So we need to convert the salary from character to integer. For this purpose we can use *atoi()* function.

Sample input file:

```
Aamir  12000
Amara 15000
Adnan 13000
Afzal   11500
```

Complete code of the program:

```
/*
* This program reads name and salary from a txt file
* Calculate the salaries and write the total in an output file
*/

#include <iostream.h>
#include <fstream.h>
#include <cstring>
#include <cstdlib>

main()
{
   ifstream inFile;                          // Handle for the input file
   char inputFileName[] = "salin.txt";     // file name, this file is in the current directory
   ofstream outFile;                         // Handle for the output file
   char outputFileName[] = "salout.txt"; // file name, this file is in the current directory
   const int MAX_CHAR_TO_READ = 100;     // maximum character to read in one line
   char completeLineText[MAX_CHAR_TO_READ];  // used in getLine function
   char *tokenPtr;                           // Used to get the token of a string
   int salary, totalSalary;

   salary = 0;
   totalSalary = 0;

   inFile.open(inputFileName);           // Opening the input file
   outFile.open(outputFileName);         // Opening the output file

   // Checking that file is successfully opened or not
   if (!inFile)
   {
          cout << "Can't open input file named " << inputFileName << endl;
          exit(1);
   }
   if (!outFile)
   {
          cout << "Can't open output file named " << outputFileName << endl;
```

vubookshoppk.com

```
            exit(1);
    }

    // Reading the complete file line by line and calculating the total salary
    while (!inFile.eof())
    {
        inFile.getline(completeLineText, MAX_CHAR_TO_READ);
        tokenPtr = strtok(completeLineText, " ");       // First token is name
        tokenPtr = strtok(NULL, " ");                   // 2nd token is salary

        salary = atoi(tokenPtr);
        totalSalary += salary;
    }
    // Writing the total into the output file
    outFile << "The total salary = " << totalSalary;

    // closing the files
    inFile.close();
    outFile.close();
}
```

The contents of output file:

```
The total salary = 51500
```

Exercise:
Modify the above program such that the input and output files are given as the command line arguments. Add another information in the input file i.e. the age of the employee. Calculate the average age of the employees and write it in the out put file. Write a program, which reads an input file. The structure of the input file is First Name, Middle Initial, Last Name. Create an output file with the structure First Name, Login Name, Password. First name is same as in the input file. The login name is middle initial and last name together. The password is the first four digits of the first name. First name, middle initial and last name does not contain space.

The sample input file is:

```
Syed N Ali
Muhammad A Butt
Faisal A Malik
Muhammad A Jamil
```

If the above file is used as input, the output should be as follows:

```
Syed Nali Syed
Muhammad Abutt Muha
Faisal Amalik Fais
Muhammad Ajamil Muha
```

Tips

Always close the file with the close function.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Open a file explicitly with open function
Always apply the error checking mechanism while handling with files.

vubookshoppk.com

Lecture No. 19

Reading Material

Deitel & Deitel - C++ How to Program Chapter 6

Summary

- Sequential Access Files (Continued)
- Last Lecture Reviewed
- Random Access Files
  - Position in a File
  - Setting the Position
  - Example of seekg() and tellg() Functions
  - Example of Data Insertion in the Middle of a File
  - Efficient Way of Reading and Writing Files
  - Copying a File in Reverse Order
- Sample Program 1
- Sample Program 2
- Exercises
- Tips

vubookshoppk.com

Sequential Access Files (Continued)

In the last lecture, we discussed little bit about Sequential Access Files under the topic of File Handling. Sequential Access Files are simple character files. What does the concept of sequential access mean? While working with the sequential access files, we write in a sequence, not in a random manner. A similar method is adopted while reading such a file.

In today's lecture, we will discuss both the topics of File Handling and Random Access Files.

Before going ahead, it is better to recap of File Handling discussed in the previous lecture. Let's refresh the functions or properties of File streams in our minds.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Last Lecture Reviewed

It is so far clear to all of us that we use **open ()** function to open files. Similarly, to close a file **close ()** function is used. **open ()** has some parameters in its parenthesis as **open (filename, mode)** but **close ()** brackets remain empty as it does not have any parameter.
A file can be opened for reading by using the **open ()** function. It can also be opened for writing with the help of the same **open ()** function . But different argument value will be needed for this purpose. If we are opening for reading with **ifstream** (Input File Stream), a simple provision of the filename is sufficient enough, as the default mode is for reading or input. We can also provide an additional argument like **open ("filename", ios::in)**. But this is not mandatory due to the default behavior of **ifstream**.
However, for **ofstream** (Output File Stream), we have several alternatives. If we open a file for writing, there is default mode available to destroy (delete) the previous contents of the file, therefore, we have to be careful here. On the other hand, if we don't want to destroy the contents, we can open the file in append mode (ios::app). **ios:trunc** value causes the contents of the preexisting file by the same name to be destroyed and the file is truncated to 0 length.

```
/* Following program writes an integer, a floating-point value, and
a character to a file called 'test' */

#include <iostream.h>
#include <fstream.h>

main(void)
{
        ofstream  out("test");   //Open in default output mode

        if ( !out )
        {
                cout  <<  "Cannot open file";
                return  1;
        }
        out  <<  100  <<  "   "  << 123.12  <<  "a";

        out.close();

        return  0;
}
```

If you want to open a file for writing at random positions forward and backward, the qualifier used is **ios:ate**. In this case, the file is at first opened and positioned at the end. After that, anything written to the file is appended

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

at the end. We will discuss how to move forward or backward for writing in the file later in this lecture.

```
/* Following program reads an integer, a float and a character from
the file created by the preceding program. */

#include  <iostream.h>
#include  <fstream.h>

main(void)
{
        char  ch;
        int  i;
        float  f;

        ifstream  in("test"); //Open in default output mode

        if( !in )
        {
                cout  <<  "Cannot open file";
                return  1;
        }

        in  >>  i;
        in  >>  f;
        /* Note that white spaces are being ignored, you can turn
        this off using unsetf(ios::skipws) */
        in  >>  ch;

        cout  <<  i  <<  "  "  <<  f  <<  "  "  <<  ch ;

        in.close();
        return  0;
}
```

Besides **open()** and **close ()** functions, we have also discussed how to read and write files. One way was character by character. This means if we read (get) from a file; one character is read at a time. Similarly, if we write (put), one character is written to the file. Character can be interpreted as a byte here. On the other hand, the behavior of stream extraction operator (>>) and stream insertion operator (<<) is also valid as we just saw in the simple programs above. We will discuss this a little more with few new properties in this lecture.

```
/* Code snippet to copy the file 'thisFile' to the file 'thatFile' */

ifstream  fromFile("thisFile");
```

vubookshoppk.com

```
if (!fromFile)
{
        cout  <<  "unable to open 'thisFile' for input";
}
ofstream toFile ("thatFile");
if ( !toFile )
{
        cout  <<  "unable to open 'thatFile' for output";
}
char  c ;
while (toFile  &&  fromFile.get(c))
{
         toFile.put(c);
}
```

This code:

- Creates an **ifstream** object called fromFile with a default mode of **ios::in** and connects it to **thisFile**. It opens **thisFile**.
- Checks the error state of the new **ifstream** object and, if it is in a failed state, displays the error message on the screen.
- Creates an **ofstream** object called toFile with a default mode of **ios::out** and connects it to thatFile.
- Checks the error state of toFile as above.
- Creates a char variable to hold the data while it is passed.
- Copies the contents of fromFile to toFile one character at a time.

It is, of course, undesirable to copy a file this way, one character at a time. This code is provided just as an example of using fstreams.

We have also discussed a function **getline ()**, used to read (get) one line at a time. You have to provide how many characters to read and what is the delimiter. Because this function treats the lines as character strings. If you use it to read 10 characters, it will read 9 characters from the line and add null character ('\0') at the end itself.

You are required to experiment with these functions in order to understand them completely.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Random Access Files

Now we will discuss how to access files randomly, forward and backward. Before moving forward or backward within a file, one important factor is the current position inside the file. Therefore, we must understand that there is a concept of file position (or position inside a file) i.e. a pointer into the file. While reading from and writing into a file, we should be very clear from where (which location inside the file) our process of reading or writing will start. To determine this file pointer position inside a file, we have two functions **tellg()** and **tellp()**.

Position in a File

Let's say we have opened a file stream **myfile** for reading (getting), **myfile.tellg ()** gives us the current get position of the file pointer. It returns a whole number of type **long**, which is the position of the next character to be read from that file. Similarly, **tellp ()** function is used to determine the next position to write a character while writing into a file. It also returns a **long** number.

For example, given an **fstream** object aFile:

```
Streampos original  =  aFile.tellp(); //save current position
aFile.seekp(0, ios::end);             //reposition to end of file
aFile  <<  x;                         //write a value to file
aFile.seekp(original);                //return to original position
```

So **tellg ()** and **tellp ()** are the two very useful functions while reading from or writing into the files at some certain positions.

Setting the Position

The next thing to learn is how can we position into a file or in other words how can we move forward and backward within a file. Suppose we want to open a file and start reading from 100th character. For this, we use **seekg ()** and **seekp ()** functions. Here **seekg ()** takes us to a certain position to start reading from while **seekp ()** leads to a position to write into. These functions **seekg ()** and **seekp ()** requires an argument of type **long** to let them how many bytes to move forward or backward. Whether we want to move from the beginning of a file, current position or the end of the file, this move forward or backward operation, is always relative to some position.. From the end of the file, we can only move in the backward direction. By using positive value, we tell these functions to move in the forward direction .Likewise, we intend to move in the backward direction by providing a negative number.

By writing:

```
aFile. seekg (10L, ios::beg)
```

We are asking to move 10 bytes forward from the begining of the file.

vubookshoppk.com

Similarly, by writing:

```
aFile. seekg (20L, ios::cur)
```

We are moving 20 bytes in the forward direction starting from the current position. Remember, the current position can be obtained using the **tellg ()** function.

By writing:

```
aFile. seekg (-10L, ios:cur)
```

The file pointer will move 10 bytes in the backward direction from the current position. With **seekg (-100L, ios::end)**, we are moving in the backward direction by 100 bytes starting from the end of the file. We can only move in the forward direction from the beginning of the file and backward from the end of the file.

You are required to write a program to read from a file, try to move the file pointer beyond the end of file and before the beginning of the file and observe the behavior to understand it properly.

seekg() and tellg() Functions

One of the useful things we can do by employing these functions is to determine the length of the file. Think about it, how can we do it.

In the previous lectures, we have discussed **strlen ()** function that gives the number of characters inside a string. This function can also be used to determine the length of the string placed inside an array. That will give us the number of characters inside the string instead of the array length. As you already know that the length of the array can be longer than the length of the string inside it. For example, if we declare an array of 100 characters but store "Welcome to VU" string in it, the length of the string is definitely smaller than the actual size of the array and some of the space of the array is unused.

Similarly in case of files, the space occupied by a file (file size) can be more than the actual data length of the file itself.

Why the size of the file can be greater than the actual data contained in that file? The answer is little bit off the topic yet it will be good to discuss.

As you know, the disks are electromagnetic devices. They are very slow as compared to the controlling electronic devices like Processors and RAM (Random Access Memory). If we want to perform read or write operations to the disk in character by character fashion, it will be very wasteful of computer time. Take another example. Suppose ,we want to write a file, say 53 bytes long to the disk . After writing it, the next file will start from 54th byte on the disk. Obviously, this is very wasteful operation of computer time. Moreover, it is also very complex in terms of handling file storage on the disk.

To overcome this problem, disks are divided into logical blocks (chunks or clusters) and size of one block is the minimum size to read and write to the disk. While saving a file of 53 bytes, we can't allocate exactly 53 bytes but

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

have to utilize at least one block of disk space. The remaining space of the block except first 53 bytes, goes waste. Therefore, normally the size of the file (which is in blocks) is greater than the actual data length of the file. When this file will be read from the disk, the whole chunk (block) is read instead of the actual data length.

By using **seekg ()** function, we can know the actual data length of the file. For that purpose, we will open the file and go to the end of the file by asking the **seekg ()** function to move 0 bytes from the end of the file as: **seekg (0, ios::end)**. Afterwards, (as we are on end of file position), we will call **tellg ()** to give the current position in **long** number. This number is the actual data bytes inside the file. We used **seekg ()** and **tellg ()** functions combination to determine the actual data length of a file.

```
/* This is a sample program to determine the length of a file. The program
accepts the name of the file as a command-line argument. */

#include <fstream.h>
#include <stdlib.h>

ifstream inFile;
ofstream outFile;

main(int argc, char **argv)
{

 inFile.open(argv[1]);


 if(!inFile)
 {
  cout << "Error opening file in input mode"<< endl;
 }

/*  Determine file length opening it for input */
 inFile.seekg(0, ios::end);          //Go to the end of the file
 long inSize = inFile.tellg();       //Get the file pointer position
 cout << "The length of the file (inFile) is: " << inSize;
 inFile.close();


/*  Determine file length opening it for output */
 outFile.open(argv[1], ios::app);
 if(!outFile)
 {
  cout << "Error opening file in append mode"<< endl;
 }
 outFile.seekp(0, ios::end);
 long outSize = outFile.tellp();
 cout << "\nThe length of the file (outFile) is: " << outSize;
 outFile.close();
```

vubookshoppk.com

```
}
```

Run this program to see its output that shows different results for both input and output modes. Discuss it on discussion board.

Data Insertion in the Middle of a File

The question arises why to talk about **seekg ()** and **tellg ()** functions before proceeding to our original topic of random access to files. This can be well-understood from the following example. Let's suppose, we have written a file containing names, addresses and dates of birth of all students of our class. There is a record of a student, who is from **Sukkur**. After sometime, we come to know that the same student has moved to **Rawalpindi**. So we need to update his record. But that record is lying somewhere in the middle of the file. How can we update it?

We can search **Sukkur** using **seekg ()** and **tellg ()** functions. After finding it, can we update the word **sukkur** with the **Rawalpindi**. No. It is just due to the fact that **Rawalpindi** is longer in length than the word **Sukkur** and the subsequent data of Data of Birth of the student will be overwritten. So the structure of the file is disturbed and as a result your data file will be corrupted. This is one of the issues to be taken care of while writing in the middle of a sequential file.

Let's think again what is the actual problem. The file is lying on the disk. We started reading that file and reached somewhere in the middle of the file to replace data at that position. But the data we are going to replace is shorter in length as compared to the new one. Consider how is this on the disk. We need some kind of mechanism to cut the disk, slide it further to make some space to insert the data into. But this is not practically possible.

In the times of COBOL, the Merge Method was employed to insert data into the middle of the file. The logic of Merge method is to copy all the data into a new file starting from beginning of the file to the location where we want to insert data. So its algorithm is:

- Opened the data file and a new empty file.
- Started reading the data file from beginning of it.
- Kept on copying the read data into the new file until the location we want to insert data into is reached.
- Inserted (appended) new data in the new file.
- Skipped or jumped the data in the data file that is to be overwritten or replaced.
- Copied (appended) the remaining part of the file at the end of the new file

This is the only way of inserting the data in the middle of a sequential file. After this process, you may delete the old file and rename the new file with the same name as that of the old file. This was done in the past . But now nowadays, it is used some time when the size of the data is not huge. But obviously, it is very wasteful as it takes lot of time and space in case of

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

large-sized data . The file size can be in hundred of megabytes or even in gigabytes. Suppose, you have to copy the whole file just to change one word. This is just a wasteful activity. Therefore, we must have some other mechanism so that we can randomly read and write data within a file, without causing any disturbance in the structure of file.

To achieve this objective, we have to have random access file and the structure of the file should be such that it is not disturbed in case of updations or insertions in the middle . The language C/C++ does not impose any structure on the file. The file can be English text or a binary file. Be sure that a language has nothing to do with it. For a language, a file is nothing but a stream of bytes. In our previously discussed example of students of a class, it makes lot of sense that each record of the file (a student record) occupies the same space. If the record size is different, the updations will have similar problems as discussed above. So what we need do is to make sure that size of each student data is identical in the file. And the space we have decided on is large enough such that, if we wrote **Sukkur** into it, the spaces were there at the end of it. If we want to replace **Sukkur** with **Ralwalpindi** or **Mirpur Khas**, it can fit into the same allotted space. It means that the file size remains the same and no destruction takes place,. So the constant record length is the key element in resolving that issue of insertion in the middle of the file without disturbing the structure of the file. Normally, we also keep some key (it is a database terminology) inside these files. The key is used to locate the record. Consider the example of students again. Suppose we had written student's name, say **Jamil Ahmed**, city and data of birth, what could we do to locate the student to change the student's information from **Sukkur** to **Rawalpindi**. We could have written a loop to read the names and to compare it with **Jamil Ahmed** to locate the particular record of that student to replace the city to **Rawalpindi**. But this comparison of names is expensive in terms of computation. It could be nicer to store a serial number with each record of the students. That serial number will be unique for each student. It can also be roll number or ID number of a student. So we can say that replace the city of the student with id number **43** to **Rawalpindi**. So in this case, we will also be doing comparison based on the basis of ID numbers of student. Here we have made a comparison again. But is a number-related comparison, not a string comparison. It will be even easier if the file is sorted on the basis of student id numbers which have no gaps. If the data is of 50 students, the first student's id number is 1 and last one is 50.

Let's take this example little further. Suppose the record of one student can be stored in 100 bytes. The student id field that is also contained within these 100 bytes is there in the file to uniquely identify each student's record. If we want to read the 23rd student's record (with id 23) in the file. One way is the brute force technique discussed earlier to start a loop from the beginning of the file to the required student id 23.

We have added following conditions with this file.

- Each student record takes 100 bytes
- The first ten bytes of a record are student id number. Student's name and City are 40 characters long respectively and last 10 bytes are for Date of Birth.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

- The student ids are in order (sorted) and there are no holes in student ids. If let's say there is 50 students data then the file will start with id 1 student's record and end with id 50 student's record.

After becoming aware of the above-mentioned conditions, can we find a quick way of finding the 23rd's student data? The answer is obviously yes as we know that one student's data is taking 100 bytes then 22 student's data will be 22 * 100 = 2200 bytes. The data for 23rd's student starts from 2201st byte and goes to 2300th byte. We will jump first 2200 bytes of the file using **seekg ()** function and there will be no wastage of resources as there are no loops, no if-else comparisons. After being aware of structure of a student's record, we can go to the desired position and perform update operation wherever needed. We can update the name of the student, change the name of the city and correct the data of birth etc. So **seekg ()** allows us to jump to any position in the file.

seekg() is used for input file or for reading from the file while **seekp()** is used for output during the process of writing to the file. Remember, a file opened with **ifstream** is used for input and cannot be used for output. Similarly, a file opened in output mode using **ofstream** cannot be used for input mode. But a file opened with the help of **fstream**; can be used for both purposes i.e. input and output. The qualifier **ios::in || ios::out** is passed into the **open ()** function while opening the file with **fstream** for both purposes. Why are we doing the OR '||' operation for opening the file in both the modes. You might remember that when we do OR operation (if either of the expression is true), the result becomes true. The qualifiers **ios::in || ios::out** are flags and exist in memory in the form of 0's and 1's. The input flag **ios::in** has one bit on (as 1) and output flag **ios::out** possesses another bit on. When we perform OR '||' operation to these two flags, the resultant of this expression contains the bits as on (as 1) from both of the flags. So this resultant flag bits depict that the file will be used for both input and output . We can use this technique of ORing for other qualifiers as well. Remember that it is not a case of AND. Although, we want input and output , yet we have to do OR operation **ios::in || ios::out** to achieve our desired behavior.

Lets see how can these tricks work

As discussed in the example of data updation within a file, what can happen if we know the exact things and want to replace a **q** character in a sentence? We should think of the logic first as it has always to be with logic and analysis that what would be algorithm for a problem. Lets say we wrote a sentence **This is an apple** in a file and want to change it to **This is a sample**. The length of both the sentences is same.

```
/* This program firstly writes a string into a file and then replaces
its partially. It demonstrates the use of seekp(), tellp() and write()
functions. */

#include <fstream>

int main ()
{
```

```
    long  pos;
    ofstream  outfile;
    outfile.open ("test.txt");        // Open the file
    outfile.write ("This is an apple",16);  // Write the string in the file
    pos  =  outfile.tellp();              // Get the File pointer position
    outfile.seekp (pos-7);           // Move 7 positions backward
    outfile.write (" sam",4);      // Write 4 chars in the current position
    outfile.close();                    // Close the file
    return 0;
}
```

Efficient Way of Reading and Writing Files

Let's consider another example. We know how to read a file character by character, write into another file or on the screen. If we want to write into a file after reading another file, there are already enough tools to get (read) one character from a file and put (write) into the other one. We can use **inputFile.getc ()** to get a character and **outputFile.putc ()** to write a character into a file.

As mentioned earlier, there is very inefficient way of doing things . We also know that for reading and writing to disk, processing in chunks is more efficient. Can we handle more data than a single byte or a single line? The answer is yes. We can use **read ()** and **write ()** functions for this purpose. These functions are binary functions and provided as a part of the stream functions. The term binary means that they read and write in binary mode , not in characters. We tell a location in memory to **read ()** function to write the read data and with the number of bytes to read or write. Usually, read(arrayname, number of bytes) e.g. read(a, 10).

Now depending on our computer's memory, we can have a very large data in it. It may be 64K.

You are required to write two programs:

One program will be used to get to read from a file and put to write into the other file. Prepare a simple character file using notepad or any other editor of your choice. Put some data inside and expand the size of the data in the file by using the copy-paste functions. A program can also be written to make this big-sized data file. The file size should be more than 40K preferably. Read this file using **getc ()** and write it another file using **putc ()**. Try to note down the time taken by the program. Explore the **time ()** function and find out how to use it in your program to note the processing time.

Write another program to do the same operation of copying using read and write functions. How to do it?

- First declare a character array:

```
char str[10000];
```

- Call the read function for input file.

```
myInputFile.read(str, 10000);
```

- To write this, use the write function for output file.

```
myOutputFile.write(str);
```

Here, a loop will be used to process the whole file. We will see that it is much faster due to being capable of reducing the number of calls to reading and writing functions. Instead of 10000 **getc ()** calls, we are making only one **read ()** function call. The performance is also made in physical reduced disk access (read and write). Important part of the program code is given below:

```
ifstream  fi;
ofstream  fo;
. . .
. . .
fi.open("inFilename", ios::in);      // Open the input file
fo.open("outFilename", ios::out);  // Open the output file
fi.seekg(0,ios::end);                 // Go the end of input file
j = fi.tellg();                          // Get the position
fi.seekg(0,ios::beg);                 // Go to the start of input file
for(i = 0; i < j/10000; i++)
{
     fi.read(str, 10000);             // Read 10000 bytes
     fo.write(str, 10000);            // Wrote 10000 bytes
}
fi.read(str, j-(i * 10000));          // Read the remaining bytes
fo.write(str, j-(i * 10000));         // Wrote the remaining bytes
fi.close();                           // Close the input file
fo.close();                           // Close the output file
```

The fine points in this exercise are left open to discover. Like what happens if the file length is 25199 bytes. Will our above solution work? Definitely, It will work but you have to figure out what happened and why does it work. Has the last **read ()** function call read 10000 bytes? You have to take care of few things while doing file handling of character and binary files. Remember that the size of the physical file on the disk may be quite different from the actual data length contained in the file.

Copying a File in the Reverse Order

We can also try copying a file in reverse. Suppose, we want to open a text file and write it in reverse order in a new file after reading. That means the last byte of the input file will be the first byte of the output file, second last byte of the input file will be the second byte of the output file until the first byte of the input file becomes the last byte of the output file. How will we do it?

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Open the input file. One of the ways of reading the files is to go to its end and start reading in the reverse direction byte by byte. We have already discussed , how to go to the end the file using **seekg (0, ios:end)**. By now, you will be aware that while reading, the next byte is read in the forward direction. With the use of **seekg (0, ios:end)**, we are already at end of the file. Therefore, if want to read a byte here it will not work. To read a byte, we should position file pointer one byte before the byte we are going to read. So we don't want to go to the end but one byte before it by using:

```
aFile.seekg (-1, ios::end);
```

We also know that whenever we read a byte, the file pointer automatically moves one byte forward so that it is ready to read the next byte. But in our case, after positioning, the file pointer 1 byte before the end of file and reading 1 byte has caused the file pointer to move automatically to the end of file byte and there is no further data of this file to read. What we need to do now to read the next byte (second last byte of input file) in reverse order is to move 2 positions from the end of file:

```
aFile.seekg (-2, ios::end);
```

Generically, this can also be said as moving two positions back from the current position of the file pointer. It will be ready to read the next character. This is little bit tricky but interesting. So the loop to process the whole file will run in the same fashion that after initially positioning file pointer at second last byte, it will keep on moving two positions back to read the next byte until beginning of the input file is reached. We need to determine the beginning of the file to end the process properly. You are required to workout and complete this exercise, snippet of the program is given below:

```
aFile.seekg(-1L, ios::end);
while( aFile )
{
      cout << aFile.tellg() << endl;
      aFile.get(c);
     aFile.put(c);
     aFile.seekg(-2L,ios::cur) ;
 }
```

Remember, generally, if statement is very expensive computation-wise. It takes several clock cycles. Sequential reading is fairly fast but a little bit tedious. To reach to 100th location, you have to read in sequence one by one. But if you use **seekg ()** function to go to 100th location, it is very fast as compared to the sequential reading.

As discussed, in terms of speed while doing file handling are **read ()** and **write ()** functions. The thing needed to be taken care of while using these functions is that you should have enough space in memory available. We have discussed very simple example of **read ()** and **write ()** functions earlier . But it is more complex as you see in your text books. Don't get confused, you remember we used array . Array name is a pointer to the beginning of

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

the array. Basically, the **read ()** requires the starting address in memory where to write the read information and then it requires the number of bytes to read. Generally, we avoid using magic numbers in our program. Let's say we want to write an **int** into a file, the better way is to use the **sizeof ()** function that can write an integer itself without specifying number of bytes. So our statement will be like:

```
aFile.write (&i, sizeof (i));
```

What benefits one can get out of this approach?. We don't need to know the internal representation of a type as same code will be independent of any particular compiler and portable to other systems with different internal representations. You are required to write little programs and play with this function by passing different types of variables to this function to see their sizes. One can actually know that how many bytes take the char type variable, int type variable or a double or a float type variable.

You are required to write a program to write integers into a file using the **write ()** function. Open a file and by running a loop from 0 to 99, write integer counter into the file. After writing it, open the file in notepad. See if you can find integers inside of it. You will find something totally different. Try to figure out what has happened. The clue lies in the fact that this was a binary write. It is more like the internal representation of the integers not what you see on the screen. You are required to play with it and experiment it by writing programs.

It is mandatory to try out the above example . Also experiment the use of **read ()** function to read the above written file of integers and print out the integers on the screen. Can you see correct output on the screen? Secondly, change the loop counter to start from 100 to 199, write it using **write ()** function and print it on the screen after reading it into an integer variable using **read ()** function. Does that work now? Think about it and discuss it on discussion board.

vubooks.com

Sample Program 1

```
/* This is a sample program to demonstrate the use of open(), close(), seekg(), get()
functions  and streams. It expects a file named my-File.txt in the current directory having
some data strings inside it. */

#include <fstream.h>
#include <stdlib.h>

/* Declare the stream objects  */
ifstream inFile;
ofstream scrn, prnt;

main()
{
```

```
        char inChar;
        inFile.open("my-File.txt", ios::in);   // Open the file for input
        if(!inFile)
        {
                cout << "Error opening file"<< endl;
        }
        scrn.open("CON", ios::out);   // Attach the console with the output stream

        while(inFile.get(inChar))       // Read the whole file one character at a time
        {
            scrn << inChar;             // Insert read character to the output stream
         }
       scrn.close();                    // Close the output stream

       inFile.seekg(0l, ios::beg);      // Go to the beginning of the file
       prnt.open("LPT1", ios::out); // Attach the output stream with the LPT1 port

       while(inFile.get(inChar))     // Read the whole file one character at a time
       {
            prnt << inChar;          // Insert read character to the output stream
       }
       prnt.close();                 // Close the output stream
       inFile.close();               // Close the input stream
}
```

vubookshoppk.com

Sample Program 2

```
/* This sample code demostrates the use of fstream and seekg() function. It will create a
file named my-File.txt write alphabets into it, destroys the previous contents */

#include <fstream.h>

fstream rFile;      // Declare the stream object
main()
{
        char rChar;
        /* Opened the file in both input and output modes */
        rFile.open("my-File.txt", ios::in || ios::out);
        if(!rFile)
        {
                cout << "error opening file"<< endl;
        }
        /* Run the loop for whole alphabets */
        for ( rChar ='A'; rChar <='Z'; rChar++)
        {
                rFile << rChar;       // Insert the character in the file
        }
        rFile.seekg(8l, ios::beg);    // Seek the beginning and move 8 bytes forward
        rFile >>rChar;                // Take out the character from the file
        cout << "the 8th character is  " << rChar ;

        rFile.seekg(-16l, ios::end);  // Seek the end and move 16 positions backword
        rFile >>rChar;                // Take out the character at the current position
        cout << "the 16th character from the end is  " << rChar ;

        rFile.close();                // Close the file
}
```

Exercises

1. Write a program to concatenate two files. The filenames are provide as command-line arguments. The argument file on the right (first argument) will be appended to the file on the left (second argument).

2. Write a program to read from a file, try to move the file pointer beyond the end of file and before the beginning of the file and observer the behavior.

3. Write a program **reverse** to copy a file into reverse order. The program will accept the arguments like:

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

reverse org-file.txt rev-file.txt

Use the algorithm already discussed in this lecture.

4. Write a program to write integers into a file using the **write ()** function. Open a file and by running a loop from 0 to 99, write integer counter into the file. After writing it, open the file in notepad. See if you can find integers inside it.

Tips

- Be careful for file mode before opening and performing any operation on a file.
- The concept of File Pointer is essential in order to move to the desired location in a file.
- **tellg()**, **seekg()**, **tellp()** and **seekp()** functions are used for random movement (backward and forward) in a file.
- There are some restrictions (conditions) on a file to access it randomly. Like its structure and record size should be fixed.
- Ability to move backward and forward at random positions has given significance performance to the applications.
- Binary files (binary data) can not be viewed properly inside a text editor because text editors are character based.

vubookshoppk.com

Lecture No. 20

Reading Material

Deitel & Deitel - C++ How to Program
Chapter 6, 16

6.2, 6.3, 6.4, 16.2,
16.3, 16.4, 16.5

Summary

1) Structures
   - Declaration of a Structure
   - Initializing Structures
   - Functions and structures
   - Arrays of structures
   - sizeof operator
2) Sample Program 1
3) Sample Program 2
4) Unions

vubookshoppk.com

Structures

Today, we will discuss the concepts of structures and unions which are very interesting part of C language. These are also in C++. After dilating upon structures, we will move to the concept of classes, quite similar to 'structures'.

What a structure is? We can understand 'structure' with the example of students of a class discussed in some of the earlier lectures. Suppose we have data about students of a class i.e. name, addresses, date of birth, GPA and courses of study. This information is related to only a single entity i.e. student. To understand the matter further, we can think of a car with its specifications like model, manufacturer company, number of seats and so on. But there is always a requirement in most of our data processing applications that the relevant data should be grouped and handled as a group. This is what the concept of structure is. In structure, we introduce a new data type. In the previous lectures, we had been dealing with int, float, double and char in our programs. You are fully familiar with the term 'strings' but there is no data type called strings. We have used 'array of char' as strings. While dealing with numbers, there is no built-in mechanism to handle the complex numbers. This means that there is no data type like complex. The FORTRAN language (Formula Translation) written for scientific application, has a complex data type. Therefore, in FORTRAN, we can say *complex x;* now *x* is a variable of type complex and has a real part and an imaginary part. There is no complex data type in C and C++. While trying to solve the quadratic equation on the similar grounds, we may have a complex number as answer i.e. if we have to calculate the square root of -1, an iota (i) will be used. So the

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

combination of real and imaginary parts is called complex number. In C, C++ we deal with such situations with structures. So a structure is not simply a grouping of real world data like students, car etc, it also has mathematical usage like complex number. The definition of structure is as under:

"A structure is a collection of variables under a single name. These variables can be of different types, and each has a name that is used to select it from the structure"

Declaration of a Structure:

Structures are syntactically defined with the word struct. So *struct* is another keyword that cannot be used as variable name. Followed by the name of the structure. The data, contained in the structure, is defined in the curly braces. All the variables that we have been using can be part of structure. For example:

```
struct student{
        char name[60];
        char address[100];
        float GPA;
};
```

Here we have a declared a structure, 'student' containing different elements. The name of the student is declared as char array. For the address, we have declared an array of hundred characters. To store the GPA, we defined it as float variable type. The variables which are part of structure are called data members i.e. name, address and GPA are data members of *student*. Now this is a new data type which can be written as:

```
        student std1, std2;
```

Here *std1* and *std2* are variables of type *student* like *int x, y;* x and y in this case are variables of *int* data type. This shows the power of C and C++ language and their extensibility. Moreover, it means that we can create new data types depending upon the requirements. Structures may also be defined at the time of declaration in the following manner:

```
struct student{
        char name[60];
        char address[100];
        float GPA;
}std1, std2;
```

We can give the variable names after the closing curly brace of structure declaration. These variables are in a comma-separated list.

Structures can also contain pointers which also fall under the category of data type. So we can have a pointer to something as a part of a structure. We can't have the same structure within itself but can have other structures. Let's say we have a structure of an address. It contains streetAddress like 34 muslim town, city like sukhar, rawalpindi, etc and country like Pakistan. It can be written in C language as:

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
struct address{
        char streetAddress[100];
        char city[50];
        char country[50];
}
```

Now the structure *address* can be a part of *student* structure. We can rewrite *student* structure as under:

```
struct student{
        char name[60];
        address stdAdd;
        float GPA;
};
```

Here *stdAdd* is a variable of type *Address* and a part of *student* structure. So we can have pointers and other structures in a structure. We can also have pointers to a structure in a structure. We know that pointer hold the memory address of the variable. If we have a pointer to an array, it will contain the memory address of the first element of the array. Similarly, the pointer to the structure points to the starting point where the data of the structure is stored.

We have used the card-shuffling example before. What will be the structure of card? Its one attribute may be the suit i.e. spades, clubs, diamonds or hearts. The second attribute is the value of the card like ace, deuce, 3 up to king. The structure of card contains a char pointer to suit and a char pointer to value i.e.

```
struct card {
        char *suit;
        char *value;
};
        card card1, card2;
```

We have defined *card1* and *card2* of type *card*. We can also define more cards. There are also arrays of structure. The syntax is same as with the normal data type. So a set of cards or an array of hundred students can be defined as under:

```
        card fullSet[52];
        student s[100];
```

The pointers to structure can be defined in the following manner i.e.

```
        student *sptr;
```

Here *sptr* is a pointer to a data type of structure *student*. Briefly speaking, we have defined a new data type. Using structures we can declare:
Simple variables of new structure
Pointers to structure
Arrays of structure

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

There are also limitation with structures as we can not say *card1 + card2;* As the operator plus (+) does not know how to add two structures. We will learn to overcome these limitations at the advanced stage. On the other hand, assignment of structures works. Therefore if *s1* and *s2* are of type *student* structure, we can say that *s1 = s2*. The assignment works because the structure is identical. So the *name* will be copied to the *name*, *address* to *address* and so on. If we want to display the structure with *cout*, it will also work. The *cout* is a very intelligent function as it interprets the structure besides showing the output.

Initializing Structures

We have so far learnt how to define a structure and declare its variables. Let's see how can we put the values in its data members. The following example can help us understand the phenomenon further.

```
struct student{
        char name[64];
        char course[128];
        int age;
        int year;
};

student s1, s2, s3;
```

Once the structure is defined, the variables of that structure type can be declared. Initialization may take place at the time of declaration i.e.

```
student s1 = {"Ali", "CS201", 19, 2002 };
```

In the above statement, we have declared a variable *s1* of data type *student* structure and initialize its data member. The values of data members of *s1* are comma separated in curly braces. "Ali" will be assigned to *name*, "CS201" will be assigned to the *course*, 19 to *age* and 2002 to *year*. So far we have not touched these data members directly.

To access the data members of structure, dot operator (.) is used. Therefore while manipulating *name* of *s1*, we will say ***s1.name.*** This is a way of referring to a data member of a structure. This may be written as:

```
s1.age = 20;
s1.year = 2002;
```

The above statement will assign the value 20 to the *age* data member of structure *s1*. Can we assign a string to the name of *s1*? Write programs to see how to do this? You may need string copy function to do this. Also, initialize the pointers to structure and see what is the difference.

Similarly, to get the output of data members on the screen, we use dot operator. To display the *name* of *s1* we can write it as:

```
cout << "The name of s1 = " << s1.name;
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Other data members can be displayed on the screen in the same fashion.

Here is a simple example showing the initialization and displaying the structure.

```
/*  Simple program showing the initialization of structure.*/

#include <iostream.h>

main()
{
        // Declaring student structure
        struct student{
                char name[64];
                char course[128];
                int age;
                int year;
        };
        // Initializing the structure
        student s1 = {"Ali", "CS201- Introduction to programming", 22, 2002};

        cout << "Displaying the structure data members" << endl;
        cout << "The name is " << s1.name << endl;
        cout << "The course is " << s1.course << endl;
        cout << "The age is " << s1.age << endl;
        cout << "The year is " << s1.year << endl;
}
```

The output of the above program is:

```
Displaying the structure data members
The name is Ali
The course is CS201- Introduction to programming
The age is 22
The year is 2002
```

Here, *s1* is a unit. The data members have been grouped together. If we have *s1* and *s2 as* two variables of student type and want to copy the data of *s1* to *s2*, it can be written as:

```
s2 = s1;
```

Functions and structures

We can pass structures to functions. Structures are passed into functions as per the C/C++ calling conventions by value. In other words, a copy of entire structure is put on the stack. The function is called which removes it from the stack and uses the structure. We can also pass the structures by reference to function. This can be performed in the same way we do with the normal variables i.e. pass the address of the structure to the function. This is call by reference.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

When we pass an array to a function, the reference of the array is passed to the function. Any change in the array elements changes the original array. Suppose we have a structure containing an array. What will happen to the array if the structures are passed as value? Is the array passed as value or reference? As the array is a part of structure, it will be passed as value. The advantage of 'pass by value' process is that if the function makes some changes to the array elements, it does not affect the original array. However, it may be disadvantageous as the complete array is copied on the stack and we can run out of memory space. So be careful while passing the structures to functions. We know that functions return value, int, char etc. Similarly functions can also return structures. In a way, the behavior of structure is same as ordinary data type.

Suppose we have a pointer to structure as *student *sptr;* here *sptr* is a pointer to *student*. Now *s1* is a variable of type *student* and *sptr = &s1 and sptr* is pointing to *s1*. How can we access the data with *sptr*? We cannot say **sptr.name*. The precedence of dot operator (.) is higher than * operator. So dot operator is evaluated first and then * operator. The compiler will give error on the above statement. To get the results, we have to evaluate * operator first i.e. *(*sptr).name* will give the desired result. There is another easy and short way to access the structure's data member i.e. using the arrow (->) in place of dot operator. We normally use the arrow (-> i.e. minus sign and then the greater than sign) to manipulate the structure's data with pointers. So to access the name with *sptr* we will write:

```
sptr->name;
```

Remember the difference between the access mechanism of structure while using the simple variable and pointer.

While accessing through a simple variable, use dot operator i.e. s1.name
While accessing through the pointer to structure, use arrow operator i.e. sptr->name;

Following is the example, depicting the access mechanism of structure's data member using the pointer to structure.
The code of the sample example is:

```
/* This program shows the access of structure data members with pointer to structure */
#include <iostream.h>

main()
{
        // Declaration of student structure
        struct student{
                char name[64];
                char course[128];
                int age;
                int year;
        };
        // Initializing the s1
        student s1 = {"Ali", "CS201- Introduction to programming", 22, 2002};
        student *sptr;
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
        // Assigning a structure to pointer
        sptr = &s1;

        cout << "Displaying the structure data members using pointers" << endl;
        cout << "Using the * operator" << endl;
        cout << endl;
        cout << "The name is " << (*sptr).name << endl;
        cout << "The course is " << (*sptr).course << endl;
        cout << "The age is " << (*sptr).age << endl;
        cout << "The year is " << (*sptr).year << endl;
        cout << endl;

        cout << "Using the -> operator" << endl;
        cout << endl;
        cout << "The name is " << sptr->name << endl;
        cout << "The course is " << sptr->course << endl;
        cout << "The age is " << sptr->age << endl;
        cout << "The year is " << sptr->year << endl;
}
```

The output of the program is:

```
Displaying the structure data members using pointers
Using the * operator

The name is Ali
The course is CS201- Introduction to programming
The age is 22
The year is 2002

Using the -> operator

The name is Ali
The course is CS201- Introduction to programming
The age is 22
The year is 2002
```

vubookshoppk.com

Arrays of structures

Let's discuss the arrays of structure. The declaration is similar as used to deal with the simple variables. The declaration of array of hundred students is as follows:

```
student s[100];
```

In the above statement, *s* is an array of type *student* structure. The size of the array is hundred and the index will be from 0 to 99. If we have to access the *name* of first student, the first element of the array will be as under:

```
s[0].name;
```

Here *s* is the array so the index belongs to *s*. Therefore the first student is s[0], the 2nd student is s[1] and so on. To access the data members of the structure, the dot operator is used. Remember that the array index is used with the array name and not with the data member of the structure.

Sizeof operator

As discussed earlier, the sizeof operator is used to determine the size of data type. The sizeof operator can also be used with the structure. Structure contains different data types. How can we determine its size in the memory? Consider the student structure that contains two char arrays and two int data types. We can simply use the sizeof operator to determine its size. It will tell us how many bytes the structure is occupying.

```
sizeof(s1);
```

We don't need to add the size of all the data members of the structure. This operator is very useful while using the *write()* function to write the structure in the file.

Here is a small example which shows the number of bytes a structure occupies in memory.

```
/*  this program shows the memory size of a structure*/

#include <iostream.h>

main()
{
        // Declaring student structure
        struct student{
                char name[64];
                char course[128];
                int age;
                int year;
        };

        student s1 = {"Ali", "CS201- Introduction to programming", 22, 2002};
        // using sizeof operator to determine the size
        cout << "The structure s1 occupies " << sizeof(s1) << " bytes in the memory";
}
```

The output of the above program is:

```
The structure s1 occupies 200 bytes in the memory
```

Let's summarize what we can do with structures:

We can define the structure
We can declare variables of that type of structure
We can declare pointers to structure

vubookshoppk.com

We can declare arrays of structure
We can take the size of structure
We can do simple assignment of two variables of the same structure type

Sample Program 1

Problem:
Suppose we have ten students in a class. The attributes of student are name, course, age and GPA. Get the data input from the user to populate the array. Calculate the average age, average GPA of the class. Find out the grade of the class and student with max GPA.

Solution:
The problem is very simple. We will declare a structure of student with name, course, age and GPA as data members. In a loop, we will get the data from the user to populate the array. Then in a loop, we will calculate the *totalAge* and *totalGPA* of the class besides determining the max GPA in that loop. Finally calculate the average age and average GPA by dividing the *totalAge* and *totalGPA* by the number of students i.e. 10. The grade of the class can be determined by the average GPA.

The complete code of the program is:

```
/* This program calculates the average age and average GPA of a class. Also determine
the grade of the class and the student with max GPA. We will use a student structure and
manipulate it to get the desired result. */

#include <iostream.h>

main()
{
        // Declaration of student structure
        struct student
        {
           char name[30];
           char course[15];
           int  age;
           float GPA;
        };

        const int noOfStudents = 10;                    // total no of students
        student students[noOfStudents];                 // array of student structure
        int totalAge, index, averageAge;
        float totalGPA, maxGPA, averageGPA;

        //  initializing the structure, getting the input from user
    for ( int i = 0; i < noOfStudents; i++ )
    {
        cout << endl;
                cout << "Enter data for Student # : " << i + 1 << endl;
             cout << "Enter the Student's Name : " ;
                cin >> students[i].name ;
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
            cout << "Enter the Student's Course : " ;
        cin >> students[i].course ;
        cout << "Enter the Student's Age : " ;
        cin >> students[i].age ;
              cout << "Enter the Student's GPA : " ;
        cin >> students[i].GPA ;
  }

   maxGPA = 0;
   // Calculating the total age, total GPA and max GPA
   for ( int j = 0; j < noOfStudents; j++ )
   {
                totalAge = totalAge + students[j].age ;
                totalGPA = totalGPA + students[j].GPA ;

                // Determining the max GPA and storing its index
                if ( students[j].GPA > maxGPA )
        {
                maxGPA = students[j].GPA;
                        index = j;
        }
}
        // Calculating the average age
averageAge = totalAge / noOfStudents ;
cout << "\n The average age is : " << averageAge << endl;

        // Calculating the average GPA
averageGPA = totalGPA / noOfStudents ;
cout << "\n The average GPA is : " << averageGPA << endl;
cout << "\n Student with max GPA is : " << students[index].name << endl ;

// Determining the Grade    of the class
if (averageGPA > 4)
{
   cout << "\n Wrong grades have been enter" << endl ;
}
else if ( averageGPA == 4)
{
        cout << "\n The average Grade of the class is : A" << endl;
}
else if ( averageGPA >= 3)
{
        cout << "\n The average Grade of the class is : B" << endl;
}
else if ( averageGPA >= 2)
{
        cout << "\n The average Grade of the class is : C" << endl;
}
else
{
```

vubookshoppk.com

```
        cout << "\n The average Grade of the class is : F" << endl;
    }
}
```

The output of the program with three students i.e. when noOfStudents = 3

```
Enter data for Student # : 1
Enter the Student's Name : Ali
Enter the Student's Course : CS201
Enter the Student's Age : 24
Enter the Student's GPA : 3.5

Enter data for Student # : 2
Enter the Student's Name : Faisal
Enter the Student's Course : CS201
Enter the Student's Age : 22
Enter the Student's GPA : 3.6

Enter data for Student # : 3
Enter the Student's Name : Jamil
Enter the Student's Course : CS201
Enter the Student's Age : 25
Enter the Student's GPA : 3.3

 The average age is : 24

 The average GPA is : 3.46667

 Student with max GPA is : Faisal

 The average Grade of the class is : B
```

vubookshoppk.com

Sample Program 2

Problem:

Read the student data from a file, populate the structure and write the structure in another file.

Solution:

We have to read from a file. We will write a function which will read from a file and return a structure to the calling program. The prototype of function is:

```
returnType functionName (argument list)
```

As the function is returning a student structure so the return type will be '*student*'. We can name the function as getData() as it is reading from a file a returning the data (i.e. student structure). In the arguments, we can give it the handle of the file from which the data is to be read. For the simplicity, we keep the argument list empty. Therefore, the prototype of our function is as under:

student getData();

This function is going to read from a file. The handle of the file has to be global so that function can access that file handle. We will define the handle of the file before *main* function to make it global. We will open the file in the *main* function before calling the *getData* function. The *getData* function returns a student structure .We will assign this to some variable of type student like:

s1 = getData();

Where *s1* is a variable of type *student*. It means that the structure, returned by the *getData* function is assigned to *s1*. The *getData* function can read the data from the file using extraction operators (i.e. >>). As this function is returning a student structure, we declare *tmpStd* of type *student*. Data read from the file will be assigned to the *tmpStd*. In the end of the *getData* function, we will return the *tmpStd* using the return keyword (i.e. return tmpStd).

Let's have a look what is happening in the memory. When we entered into the *getData* function from main, it creates locally a *tmpStd* of type student structure. *tmpStd* is created somewhere in the memory. It starts reading data from the file assigning it at that memory location. On its return, the function copies this *tmpStd* on to the stack. Stack is the way the function communicates with the main function or calling program. When the function returns, it will destroy the *tmpStd* as it is local variable of *getData* function. It does not exist anymore. It just came into being while you were inside the *getData* function. It disappears once *getData* finishes. However, before it disappears, the *getData* copies *tmpStd* in the memory so the main function pick up those value use it to assign to *s1*. Similarly we write the *writeData* function to write the data into a file. We will pass this function a student type structure to write it on the file. The prototype of *writeData* is as:

void writeData(student s1);

The sample input file:

```
nasir
CS201
23
3
Jamil
CS201
31
4
Faisal
CS201
25
3.5
```

Here is the complete code of the program:

```
/* this program reads from a file, populate the structure, and write the structure in a file */

#include <stdlib.h>
```

```
#include <fstream.h>

// Global variables for input and output files
ifstream inFile;
ofstream outFile;

//student structure
struct student
{
   char name[30];
   char course[15];
   int  age;
   float GPA;
};

// function declarations
void openFile();                    // open the input and output files
student getData();                  // Read the data from the file
void writeData(student);        // write the structure into a file

void main()
{
    const int noOfStudents = 3;                 // Total no of students
    openFile();                                 // opening input and output files
    student students[noOfStudents];  // array of students

        // Reading the data from the file and populating the array
    for(int i = 0; i < noOfStudents; i++)
    {
       if (!inFile.eof())
       {
          students[i] = getData();
       }
       else
       {
          break ;
       }
    }

        // Writing the structures to the file
    for(int i = 0; i < noOfStudents; i++)
    {
        writeData(students[i]);
    }

        // Closing the input and output files
    inFile.close ( ) ;
    outFile.close ( ) ;
}
```

vubookshoppk.com

```
/* This function opens the input file and output file */
void openFile()
{
    inFile.open("SAMPLE.TXT", ios::in);
    inFile.seekg(0L, ios::beg);
    outFile.open("SAMPLEOUT.TXT", ios::out | ios::app);
    outFile.seekp(0L, ios::end);

    if(!inFile || !outFile)
    {
      cout << "Error in opening the file" << endl;
          exit(1);
    }
}

/* This function reads from the file */
student getData()
{
    student tempStudent;
          // temp variables for reading the data from file
    char tempAge[2];
    char tempGPA[5];

    // Reading a line from the file and assigning to the variables
    inFile.getline(tempStudent.name, '\n');
    inFile.getline(tempStudent.course, '\n');
    inFile.getline(tempAge, '\n');
    tempStudent.age = atoi(tempAge);
    inFile.getline(tempGPA, '\n');
    tempStudent.GPA = atof(tempGPA);

    // Returning the tempStudent structure
    return tempStudent;
}

/* This function writes into the file the student structure*/
void writeData(student writeStudent)
{
    outFile << writeStudent.name << endl;
    outFile << writeStudent.course << endl;
    outFile << writeStudent.age << endl;
    outFile << writeStudent.GPA << endl;
}
```

The contents of output file is:

```
nasir
CS201
23
3
```

vubookshoppk.com

```
Jamil
CS201
31
4
Faisal
CS201
25
3.5
```

Unions

We have another construct named union. The concept of union in C/C++ is: if we have something in the memory, is there only one way to access that memory location or there are other ways to access it. We have been using int and char interchangeably in our programs. We have already developed a program that prints the ACSII codes. In this program, we have stored a char inside an integer. Is it possible to have a memory location and use it as int or char interchangeably? For such purposes, the construct union is used. The syntax of union is:

```
union intOrChar{
        int i,
        char c;
};
```

The syntax is similar as that of structure. In structures, we have different data members and all of these have their own memory space. In union, the memory location is same while the first data member is one name for that memory location. However, the 2nd data member is another name for the same location and so on. Consider the above union (i.e. intOrChar) that contains an integer and a character as data members. What will be the size of this union? The answer is the very simple. The union will be allocated the memory equal to that of the largest size data member. If the int occupies four bytes on our system and char occupies one byte, the union *intOrChar* will occupy four bytes. Consider another example:

```
union intOrDouble{
        int ival;
        double dval;
};
```

The above union has two data members i.e. *ival* of type int and *dval* of type double. We know that double occupies more memory space than integer. Therefore, the union will occupy the memory space equivalent to double. The data members of unions are accessed in a similar way as we use with structures i.e. using the dot operator. For example:

```
intOrDouble uval;
uval.ival = 10;
```

To get the output of the data members, cout can be used as:

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

cout << " The value in ival = " << uval.ival;

It will print "The value in ival = 10". Now what will be output of the following statement?

cout << " The value in dval = " << uval.dval;

We don't know. The reason is that in the eight bytes of double, integer is written somewhere. When we use integer, it is printed fine. When we printed the double, the value of int will not be displayed. Rather something else will be printed. Similarly in the following statement i.e.

```
uval.dval = 100.0;
cout << " The value in dval = " << uval.dval;
```

It will print the right value of *dval*. The value of this double is written in such a way that it will not be interpreted by the integer. If we try to print out *ival*, it will not display 100. Unions are little bit safer for integer and characters. But we have to think in terms that where to store the value in memory.

Suppose, we have some integer value 123 and want to append 456 to it so that it becomes 123456. How can we do that? To obtain this result, we have to shift the integer three decimal places i.e. we can multiply the integer 123 by 1000 (i.e. 123000) and then add 456 to it (i.e. 123456). Consider a union containing four characters and an integer. Now the size of the char is one and integer is four so the size of the union will be four. We assign the character 'a' to the integer, and display the chars and integer value. If we want to shift the value of first byte into the second byte, the integer will be multiplied by 256(i.e. A byte contains 8 bits and 2 to power 8 is 256), then add character 'b' to it. We see that the char variables of union contains 'a' and 'b'.

Here is the code of the program;

```
/* This program uses a union of int and char and display the memory usage by both */
#include <iostream.h>

main()
{
   // Declaration of union
   union intOrChar{
      char c[4];
      int x;
   }u1;

   u1.x  = 'a';              // Assigning 'a' to x
   // Displaying the char array and integer value
   cout << "The value of c = " << u1.c[0] << "," << u1.c[1]
        << "," << u1.c[2] << "," << u1.c[3]<< endl;
   cout << "The value of x = " << u1.x << endl;

   // Shifting the values one byte and adding 'b' to the int
```

```
    u1.x *= 256;
    u1.x += 'b';

    // Displaying the char array and integer value
    cout << "The value of c = " << u1.c[0] << "," << u1.c[1]
         << "," << u1.c[2] << "," << u1.c[3]<< endl;
    cout << "The value of x = " << u1.x << endl;

    // Shifting the values one byte and adding ‘b’ to the int
    u1.x *= 256;
    u1.x += 'c';

    // Displaying the char array and integer value
    cout << "The value of c = " << u1.c[0] << "," << u1.c[1]
         << "," << u1.c[2] << "," << u1.c[3]<< endl;
    cout << "The value of x = " << u1.x << endl;

    // Shifting the values one byte and adding ‘b’ to the int
    u1.x *= 256;
    u1.x += 'd';

    // Displaying the char array and integer value
    cout << "The value of c = " << u1.c[0] << "," << u1.c[1]
         << "," << u1.c[2] << "," << u1.c[3]<< endl;
    cout << "The value of x = " << u1.x << endl;
}
```

The output of the program is;

```
The value of c = a, , ,
The value of x = 97
The value of c = b,a, ,
The value of x = 24930
The value of c = c,b,a,
The value of x = 6382179
The value of c = d,c,b,a
The value of x = 1633837924
```

Unions are very rarely used. They become very important when we want to do some super efficient programming. Experiment with the unions and structures.

We have learnt how to use structures and unions. These are relatively less used parts of C/C++ language. But structures at least are very useful. They allow us a convenient way of grouping data about a single entity. We have used student entity in our example. You can think of a car or any other object and find out its properties before grouping them in a structure. We don’t need to manipulate its properties individually as grouping them into a unit is a better option. Try to write different programs using structures.

vubookshoppk.com

Lecture No. 21

Reading Material

Deitel & Deitel - C++ How to Program Chapter. 16

16.7

Summary

Bit Manipulation
Bit Manipulation Operators
AND Operator
OR Operator
Exclusive OR Operator
NOT Operator
Bit Flags
Masking
Unsigned Integers
Sample Program
Shift Operators

Bit Manipulation

We have so far been dealing with bytes using different data types. In this lecture, we will see what a bit is? Bit is the basic unit of memory. Eight bits form a byte. As you know that data is stored in computers in 0's and 1's form. An integer uses four bytes and the integer calculations occur in four bytes. Thus, we are manipulating bytes while using different data types. Now we will try to understand the process of 'bit manipulation'. Now we will deal with each bit in a byte and explore how to do on or off each bit. A bit, having 1 is said on while the one with 0 is called off. Here we will discuss different operators to manipulate bits.

The concept of bit manipulation means that we can do work with a bit, the smallest unit of memory. Bit manipulations utilize very small memory. Thus, we can make an efficient use of the memory. The bit fields are of great use in operating systems and files attributes. The bit manipulations are useful while working at operating system level.

Let's have a look on different operators, used for bit manipulations.

Bit Manipulation Operators

The following table shows different operators used for bit manipulation.

Operator	**Operator Sign**
Bitwise **AND** Operator	&
Bitwise **OR** Operator	|
Bitwise Exclusive OR Operator	^
NOT Operator	~
Left Shift Operator	<<

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Right Shift Operator	>>

Here & is the bit-wise AND operator. Don't confuse it with the logical AND operator &&. Similarly | is the bit-wise OR operator. Don't confuse it with the logical OR operator ||.

Now let's talk about these operators in detail.

AND Operator (&)
The AND operator (&) works just like the logical AND operator (&&) but on bits. It compares two bits and returns 1 if both bits are 1. If any of the two bits being compared is 0, the result will be 0.

Following table, also called truth table, will further explain the operation of & operator.

Bit1	**Bit2**	**Bit1 & Bit2**
1	1	1
1	0	0
0	1	0
0	0	0

We know that when a number is stored in memory, it gets stored in bit pattern which has binary representation (only 1 and 0). So we can use & to AND two numbers bit-wise. To understand this, consider the following example.
Suppose we have two numbers - 12 and 8 and want to apply & on these ones. Here we will make use of the binary number system. The binary representation (base 2 system) of 12 and 8 are as 12 = $(1100)_2$ and 8 = $(1000)_2$. Now we apply the & operator on these numbers and get the result as follows

```
12 =    1       1       0       0
&
                                8 =    1       0       0       0
                               ------------------------------
                                       1       0       0       0
```

Thus 12 & 8 = $(1000)_2$ = 8. Don't think 12 & 8 as an arithmetic operation. It is just a bit manipulation or a pattern matching issue. Each bit of first number is matched (compared) with corresponding bit of the second number. The result of & is 1 if both bits are 1. Otherwise, it will be 0. The & operator is different from the && operator. The && operator operates on two conditions (expressions) and returns true or false while the & operator works on bits (or bit pattern) and returns a bit (or bit pattern) in 1 or 0.

Example 1
We want to determine whether in a number a specific bit is 1 or 0. Suppose we want to determine whether the fourth bit (i.e. 2^3) of a number is 1 or 0. We will pick the number whose fourth bit is 1 and the remaining are zero. It is 2^3 (i.e. 8). Now we will take AND of the given number with 8 (i.e 1000 in bit pattern.). In bit manipulation, the number is written in hexadecimal form. In the C language, we put 0x or 0X before

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

the number to write a number in hexadecimal. Here we will write 8 as 0x8 in our code. Now all the bits of 8 are zero except the fourth one which is 1. The result of the number being ANDed with 8 will be non-zero if the fourth bit of the number is 1. As the fourth bit of 8 is also 1, & of these two bits will result 1. We call the result non-zero just due to the fact that we are not concerned with the numbers like 1,2,3 or whatsoever. We will write this in the form of a statement as under

if (number & 0x8)

instead of **if ((number & ox8) > =1)**

The **if** looks for a true or false. Any non-zero value is considered true and a zero is considered false. When we do bit-wise AND of two numbers if the result is non-zero (not 1 only, it may be 1 or any other number), this **if** statement will be true. Otherwise, it will be false.

By a non-zero value we simply conclude that the fourth bit of the number is set (i.e. 1). A bit is said to be set in case it is 1 and 'not set' if it is 0. This way, we can set any bit pattern in the power of 2, to determine whether a specific bit of a number is set or not. For example, to determine bit no. 3 of a number we can AND it with 2^2 (4).

Following is the code of the example finding out whether the fourth bit of a number is set (1) or not set (0).

```
//This program determines whether the fourth bit of a number entered by user is set or not

#include <iostream.h>
main()
{
        int number ;
        cout << "Please enter a number  " ;
        cin >> number ;
        if (number & 0x8 )      //8 is written in hexadecimal form
                cout << "The fourth bit of the number is set" << endl;
        else
                cout << "The fourth bit of the number is not set" << endl;
}
```

Sample output of the program.

```
Please enter a number  12
The fourth bit of the number is set
```

OR Operator (|)

The OR operator, represented by '|' works just like the & operator with the only difference that it returns 1 if any one of the bits is 1. In other words, it returns 0 only if both the input bits are 0. The | (bit-wise OR) operator is different from the || (logical OR) operator. The || operator operates on two conditions (expressions) and returns true or false while the | operator works on bits (bit pattern) and returns a bit (or bit pattern) in 1 or 0.

The truth table of OR operator is given below.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Bit1	Bit2	Bit1 | Bit2
1	1	1
1	0	1
0	1	1
0	0	0

We can make it sure that a specific bit in a number should be 1 with the help of | operator. For this purpose, we take OR of this number with another number whose bit pattern has 1 in that specific bit. Then OR will produce 1 as the bit at that position in second number is 1 and OR gives 1 if any one bit is one. Thus in the output that specific bit will have 1.
Let us consider the following example in which we apply OR operator on two numbers 12 and 8.

```
12 =    1       1       0       0
|
 8 =    1       0       0       0
------------------------------
                1       1       0       0
```

Hence we get 12 | 8 = 12.
In case, x = 8 | 1, the OR operation will be as under.

```
8 =     1       0       0       0
|
1 =     0       0       0       1
-------------------------
        1       0       0       1
```

Thus x = 8 | 1 = 9.
Don't take the statement in mathematical or arithmetical terms. Rather consider it from the perspective of pattern matching.
The & operator is used to check whether a specific bit is set or not while the | operator is used to set a specific bit.

Exclusive OR Operator (^)

Exclusive OR operator uses the sign ^ . This operator returns 1 when one input is zero and the second is 1. It returns 0 if both bits are same i.e. both are either 0 or 1. The truth table of exclusive OR, also called xor (zor) , is given below.

Bit1	Bit2	Bit1 ^ Bit2
1	1	0
1	0	1
0	1	1
0	0	0

To understand exclusive OR, let's work out exclusive OR of 8 and 1.
In the following statement, the pattern matching is shown for 8 ^ 1.

```
8 =     1       0       0       0
```

vubookshoppk.com

```
      ^
1 =   0     0     0     1
-------------------------------
      1     0     0     1
```

This shows that 8 ^ 1 = 9. If we take again exclusive OR of 9 with 1. The result will be 8 again as shown below.

```
9 =   1     0     0     1
      ^
1 =   0     0     0     1
-----------------------------
      1     0     0     0
```

While taking ^ (exclusive OR) of a number with a second number and then ^ of the result with the second number, we get the first number again. This is a strength of the ^ operator that is very useful.

NOT Operator (~)

This is a unary operator. It inverts the bits of the input number, meaning that if a bit of the input number is 1, the operator will change it to 0 and vice versa. The sign ~ is used for the NOT operator. Following is the truth table of the NOT operator.

Bit1	**~ Bit1**
1	0
0	1

Let's take NOT of the number 8. This will be as follows

```
8 =   1     0     0     0
```

Now ~8 will invert the bits from 1 to 0 and from 0 to 1. Thus ~8 will be

```
~8 =  0     1     1     1
```

which is 7.

The bit manipulation operators are very useful. Let's consider some examples to see the usefulness of these operators.

Example (Bit Flags)

The first example relates to operating system. In Windows, you can view the properties of a file. You can get the option properties by right clicking the mouse on the file name in any folder structure. You will see a window showing the properties of the file. This will show the name of the file, the date of creation/modification of the file etc. In the below part of this window, you will see some boxes with check marks. These include read only and archive etc. While looking at a check mark, you feel of having a look at a bit. If there is a check mark, it means 1. Otherwise, it will be 0. So we are looking at bit flags which will depict the status of the file. If the file is marked read-only, a specific bit is set to 1 in the operating system. This 1 indicates that the status of the file is read-only.
When we look for directory in UNIX operating system, rwx, rx or rw are seen before the name of a file. The rwx are actually symbols used for read, write and execute permissions of the file. These are the attributes of the file.

vubookshoppk.com

In operating systems, the attributes of a file are best get as bit fields. The 1 in a bit means the attribute is set and 0 means the attribute is not set (or cleared).

Example (Masking)

Let's see how ^ operator works. Whenever you log on to a system or server or to a web site like yahoo or hotmail, you enter your user name and then the password. The system or server validates your password and allows the access. Your password is kept in the database of the system/server. When you enter the password, the system compares it with the one earlier stored in its database. If it matches, the system allows you to access the system. But there may be a problem at this stage from the security perspective. If the password is stored in the database as it is, then the administrative (or any person having access to database) can read the password of any account. He can make misuse of password. To prevent this and make the password secure, most of the operating systems keep the password in an encrypted fashion. It codes the passwords to a different bit pattern before storing it in its database so that no body can read it. Now when a user enters his password, there are two methods to compare this password with the password earlier stored in the database. Under the first method, on entering the password, the password stored will be decoded to the original password and compare with the password entered. This is not a best way because of two reasons. If there is a method to decrypt a password, the administrator can decrypt the password for any sort of misuse. The second method is that when you enter a password, it travels through wires to go to somewhere for comparison. While it is traveling on wire, someone can get it. Another reason to compare the password in encrypted form is that it is very easy to do encryption but the decryption process is very difficult. Therefore, to make this process secure and easy, the password entered is encrypted and compared to the password in the database, which is already stored in encrypted form.

The Exclusive OR operator (^) can be used to encrypt and decrypt the password. Suppose there are two numbers **a** and **b**. We take **c = a ^ b**. Now if we take ^ of the result **c** with **b** (i.e. c ^ b), the result will be **a**. Similarly, if we take Exclusive OR of the result **c** with **a** (c ^ a) , the answer will be **b**. You can do exercise this phenomenon by taking any values of **a** and **b**. This phenomenon of Exclusive OR can be used to secure a password. You can take Exclusive OR of the password with a secret number and save it to the database. Now when it is needed to be compared with entered password, you again take Exclusive OR of the saved password with the same secret number and get the original password back. If someone else wants to get the password, it is very difficult for him/her to get that because the original password will be got by taking Exclusive OR of the saved password with the same secret number.

Here is another example of Exclusive OR. Sometimes, there are bad sectors in a hard disk, which bring it to a halt. We cannot access our data from it. This is worst situation. In large systems like servers, there is a requirement that these should work twenty four hours a day, seven days a week. In such systems, we cannot take the risk. To avoid this and meet the requirements, we use a technique which is called RAID. RAID stands for Redundant Array of Inexpensive Devices. In this technique, we use many disks instead of one. Suppose we have nine disks. Now when we say write a byte on the disk, The RAID will write a bit on first disk then second bit on the second disk and so on. Thus 8 bits (one byte) are written on 8 disks. Now what will be written on the ninth disk? We take exclusive OR of these 8 bits pair by pair and write the result on the ninth disk. The benefit of this process that in case one disk stops

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

working, we may place a new disk in its place. And to write a bit on this disk, we again take Exclusive OR of eight bits on the other disks and write the result on this disk. This will be the same bit that was written in the damaged disk.
You can prove it by the doing the following exercise on paper.
Write eight bits, take their Exclusive OR one by one and write it at ninth position. Now erase any one bit and take Exclusive OR of the remaining eight bits. You will get the same bit which was erased. Thus it is a useful technique for recovering the lost data without shutting down the system. We replace the bad disk with a new one while the system is on. The system using the RAID technique, writes the data to the new disk. This technique of replacing a disk is known as Hot Plug.

We have read the technique of swapping two numbers. In this method, we use a third temporary place to swap two numbers. Suppose **a** and **b** are to be swapped. We store **a** in a temporary place **c**. Then we store **b** in **a** and put the value of **c** (which has the value of **a**) in **b**. Thus **a** and **b** are swapped.
We can swap two numbers without using a third place with the help of Exclusive OR. Suppose we want to swap two unsigned numbers **a** and **b.** These can be swapped by the following three statements.

```
a = a ^ b ;
b = b ^ a ;
a = a ^ b ;
```

Do exercises of this swap technique by taking different values of **a** and **b**.

Unsigned Integers
The bit manipulations are done with unsigned integers. The most significant bit is used as a sign bit. If this bit is zero, the number is considered positive. However, if it is 1, the number will be considered negative. Normally these bit manipulations are done with unsigned integers. The unsigned integers are declared explicitly by using the word 'unsigned' as follow.

```
unsigned int i, j, k ;
```

By this declaration the integers i, j and k will be treated as positive numbers only.

Sample Program

The following program demonstrate the encryption and decryption of a password. The program takes a password from user, encrypts it by using Exclusive OR (^) with a number. It displays the encrypted password. Then it decrypts the encrypted password using Exclusive OR (^) with the same number and we get the original password again.

Following is the code of the program.

```
//This program demonstrate the encryption by using ^ operator

# include<iostream.h>
main ()
{
   char password[10] ;
   char *passptr ;
   cout << "Please enter a password(less than 10 character): " ;
   cin >> password ;
```

vubookshoppk.com

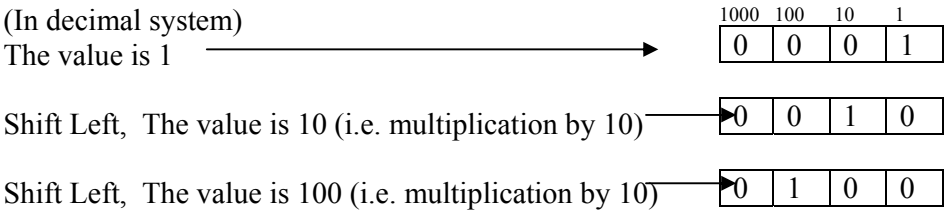
```
    passptr = password ;
    //now encrypting the password by using ^ with 3
    while (*passptr != '\0' )
    {
      *passptr =  (*passptr ^ 3);
       ++passptr ;
    }
    cout << "The encrypted password is:  " << password << endl;

    //now decrypting the encrypted password by using ^ with 3

    passptr = password ;
    while (*passptr != '\0' )
    {
      *passptr =  (*passptr ^ 3);
       ++passptr ;
    }
     cout << "The decrypted password is:  " << password << endl;
}
```

The following is a sample output of the program.

```
Please enter a password(less than 10 character): zafar123
The encrypted password is:  ybebq210
The decrypted password is:  zafar123
```

Shift Operators

Shifting the binary numbers is similar to shifting the decimal numbers. Suppose we have 1 in decimal system and want to shift it left in a way that zero is put at the ending place. Thus 1 becomes 10. Mathematically, it is a multiplication by 10. Now if we shift 10 to left and place 0 at the last place, we get 100. It is again a multiplication by 10. In pictorial terms, we can show this as under.

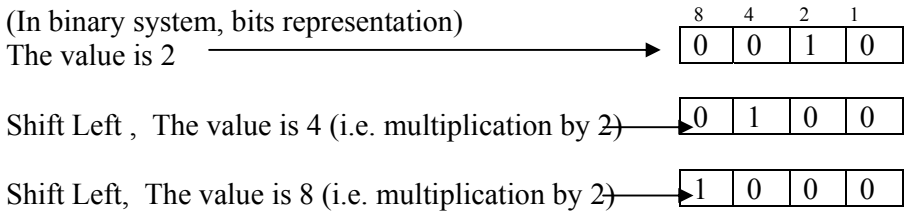


The same thing applies when we do bit shifts. If we shift a bit to the left in the binary system, it is multiplied by 2. If we do left shift again we are multiplying by 2 again. Same applies in the other direction. By shifting to the right, we will be dividing by 2 in the binary system and dividing by 10 in decimal system. In this process, the shifted digit/bit is discarded. When we do left shift, zeroes are inserted in the right side bits. The same applies to right shift, as zeros are inserted in the left side bits. But the situation will be different if we use signed numbers. As we know that in signed numbers the most significant bit is 1. Now you have to see that what happens while

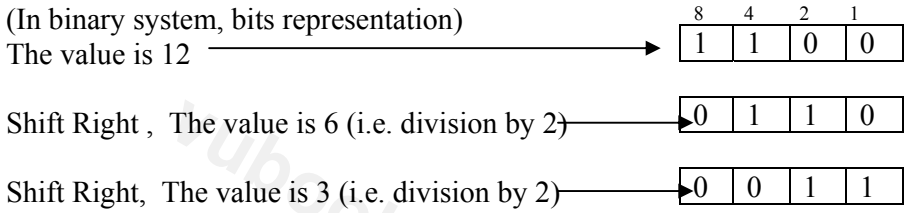
All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

right shifting the signed number? If zero is inserted at the left most bit, the negative number will become a positive number. Normally the operating systems or compilers treat it differently.
The following figures show the shift operations.

Shift Left:



Shift Right:



We have specific operators for left and right shifts. The left shift operator is << and right shift operator is >>. These are the same signs as used with *cout* and *cin*. But these are shift operators. We can give a number with these operators to carry out shift operation for that number of times. The following program demonstrates the left and right shift operators.

```
//This program demonstrate the left and right shift

# include <iostream.h>
main()
{
   int number, result ;
   cout << "Please enter a number:    " ;
   cin >> number ;
   result = number << 1 ;
   cout << "The number after left shift is    " <<  result  << endl ;
   cout << "The number after left shift again is    " << (result << 1) << endl ;
   cout << "Now applying right shift"  << endl ;
   result = number >> 1 ;
   cout << "The number after right shift is    " <<  result << endl ;
   cout << "The number after right shift again is    " << (result  >> 1) << endl ;
}
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Here is the out put of the program.

```
Please enter a number:    12
The number after left shift is    24
The number after left shift again is    48
Now applying right shift
The number after right shift is    6
The number after right shift again is    3
```

In the output, we see that the left shift operator (<<) has multiplied the number by 2 and the right shift operator (>>) has divided the number by 2. The shift operator is more efficient than direct multiplication and division.

Exercises

Write different programs to demonstrate the use of bit manipulation operators.
Write a program which takes two numbers, displays them in binary numbers and then displays the results of AND, OR and Exclusive OR of these numbers in binary numbers so that operations can be clearly understood.
Write a program which swaps two numbers without using a temporary third variable.
Write a program, which takes a password from the user, saves it to a file in encrypted form. Then allow the user to enter the password again and compare it with the stored password and show is the password valid or not.

vubookshoppk.com

Lecture No. 22

Reading Material

Deitel & Deitel - C++ How to Program ***Review previous lectures***

Summary

- Bitwise Manipulation and Assignment Operator
- Design Recipes
- Variables
- Data Types
- Operators
  - Arithmetic operators
  - Logical operators
  - Bitwise operators
- Programming Constructs
- Decisions
  - if statement
  - Nested if statement
- Loops
  - while loop
  - do-while loop
  - for loop
- switch, break and continue Statements
- Functions
  - Function Calling
  - Top-Down Methodology
- Arrays
- Pointers
- File I/O

vubookshoppk.com

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Bitwise Manipulation and Assignment Operator

Last time we discussed bitwise operators, we will continue with the elaboration of bitwise manipulation and assignment operator.

C/C++ are well constructed languages, at start we used to write:

```
a = a + 1;
```

This is used to increment the variable. Then we came to know of doing it in a different manner:

```
a += 1;
```

This is addition and assignment operation using single operator +=.

The same thing applies to bitwise operators; we have compound assignment operators for **&** (bitwise AND), **|** (bitwise OR) and ^ (bitwise exclusive OR). It is written in the same way as for the above mentioned arithmetic operators . Suppose we want to write:

```
a = a & b;
```

It can be written as:

```
a &= b;
```

Similarly for | and ^ operations we can write the statement in the following fashion.

```
a |= b;
```

and

```
a ^= b;
```

Remember, the ~ (NOT) operator is unary as it requires only one operand. Not of a variable **a** is written as: **~a**. There is no compound assignment operator available for it.

Now we will recap topics covered in the previous lectures one by one.

Design Recipe

Our problems, typically, are of real world nature, e.g., Payroll of a company. These problems are expressed in words. As a programmer we use those words to understand the problem and to come up with its possible solution.
To begin with the comprehension and resolution process, we analyze the problem and express the problem in words in reduced and brief manner. Once we have reduced it into its essence, we put some examples to formulate it. For example, if the problem is to calculate the annual net salary of employees, we can take an example for a

vubookshoppk.com

particular employee X. Later we will refine the problem, write program and test it. Finally we review it if it has met objectives. We have discussed all these steps in Design Recipe. There was a main heading in the topic, "Pay attention to the detail". Never forget this as our computers are very dump machines. They perform exactly whatever we tell them.

It is important to keep in mind that we are using C/C++ as a vehicle to understand programming concepts.

Variables

The computer memory can be thought of as pigeon holes each with an address. To store numbers or characters in computer memory, we need a mechanism to manipulate it and data types are required for different types of data. Instead of using hard coded memory addresses with data types, symbolic names are used. These symbolic names are called variables because they can contain different values at different times. For example,

```
int  i;
double  interest;
```

i and **interest** are symbolic names or variables with types of **int** and **double** respectively.

Data Types

int type is used to store whole numbers. There are some varieties of data types to store whole numbers e.g., **short** and **long**. **unsigned** qualifier is used for non-negative numbers. To represent real numbers we use **float** data type. For bigger-sized real numbers **double** data type is used. **char** data type is used to store one character. Generally, the size of the **int** type on our machines is 4 bytes and **char** is 1 byte. **char**s are enclosed in single quotation mark. ASCII table contains the numeric values for **char**s.

We further discussed a bit later stage about the aggregations or collections of basic data types (**int**, **float** and **char** etc**)** called **arrays.** Arrays are used to aggregate variables of same data type.

Operators

We discussed three types of operators:

- Arithmetic Operators
- Logical Operators
- Bitwise Operators

vubookshoppk.com

Arithmetic Operators

+ operator is used to add two numbers, **-** is used to subtract one number from the other, ***** is used to multiply two numbers, / is used to divide numbers. We also have a modulus operator **%** , used to get the remainder. For example, in the following statement:

```
c = 7 % 2;
```

7 will be divided by 2 and the remainder **1** will be the stored in the variable **c**. We also used this operator in our programs where we wanted to determine evenness or oddness of a number. There are also compound arithmetic operators +=, **-=**, ***=**, /= and also **%=** for our short hand. It is pertinent to note that there is no space between these compound operators.

Logical Operators

The result for logical operators is always true or false. **&&** (AND operator) and || (OR operator). Logical Comparison operators are used to compare two numbers. These operators are: <, <=, ==, >, >=. Don't confuse the == operator of equality with = operator of assignment.

It is important for us to remember the difference between these two operators of equality (==) and assignment (=) . However, C/C++ creates a little problem for us here. When we write a statement as:

```
a = b;
```

The assignment statement itself has a value, which is the same as that of the expression on the right hand side of the assignment operator. We can recall from our last lecture that we only wrote a number inside the if statement. We also know that if the resultant inside the if statement is non-zero then its code block is executed. In case, the result is zero, the control is transferred to the else part.

If we want to compare two variables **a** and **b** inside if statement but wrongly write as:

```
if ( a = b )
{       // if code block

        // do something
}
else
{
        // do something else
}
```

In this case, if the value of the variable **b** is non-zero (and hence value of the statement **a = b** is non-zero) then if code block will be executed. But this was not required, it is a logical fault and compiler was unable to detect it. Our objective was to

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

compare two variables. For that purpose, we should have used assignment operator == for that as:

```
if ( a == b )
```

One should be very careful while using comparison operators. You should not miss any case of it and be sure about what you wanted to do and what will be the output of a comparison statement.

You should keep in mind straight line of Calculus for the sake of completeness; you should always divide your domain into two regions. If we take >= as one region then the other region is <. Similarly if we say < as a region**,** the other region is >=. Depending on the problem requirements, these regions should be very clear.

Bitwise Operators

& is bitwise AND operator, | is bitwise OR operator, ^ is bitwise Exclusive OR operator and ~ is bitwise inversion or NOT operator. ~ (NOT operator) is unary operator as it requires one operator and the remaining operators &, | and ^ are binary operators because they require two operands.

vubookshoppk.com

Programming Constructs

For us, it is not necessary to know who is the one to devise or decide about these constructs to be part of the program logic. The important thing is the concept of programming constructs, required to write a program. We have earlier discussed three constructs.

1. The **sequential execution** of statements of a program. Execution of statements begins from very first statement and goes on to the last statement.

2. Secondly we need **decisions** that if something is true then we need to do something otherwise we will do something else. We use if statement for this.

3. The third construct is **loops**. Loops are employed for repetitive structures.

Decisions

Normally, if statement is used where decisions are required.

If statement

The syntax of if statement is fairly simple i.e.

```
if (condition)
{
        // if code block
```

```
}
else
{
        // else code block
}
```

The result of the **condition** can be either true or false. If the condition is true, **if code block** is executed. Braces of the if code block are mandatory but if there is only one statement in the if code block then the braces can be omitted or are optional. Now if the **condition** is false, the if code block is skipped and the control is transferred to the else part and else code block is executed**. Else part is optional** to associate with the if part. So without else the statement looks like the following:

```
if (condition)
{       // if code block
        // Do something here
}
```

Use of braces is again mandatory. Again, however, if there is only statement inside the else part then brace is optional.
As a programming practice, use of braces all the time is recommended. It makes your program more readable and logically sound.

What happens when the condition is complex?

Nested if statement

For complex conditions, we use logical connectives like **&&**, ||. For example:

```
if ( a > b && a < c)
```

If there are **nested decisions** structure that we want to do something based on some condition and further we want to do something more based on an additional condition. Then we use nested if-statements as under:

```
if ( a > b && a < c )
{
        // Do something

        if ( a == 100 )
        {
                // Do something more
        }
        else
        {
                // Do something else more
        }
}
else
{
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
        // Do something else
}
```

From stylistic and readability perspective, we properly indent the statements inside if-statements as shown above.
We discussed pictorial representation of if-statement. By using flowchart of if statement that was a bit different than normally we see inside books, we introduced structured flowcharting.
In **structured flowcharting**, we never go to the left of the straight line that joins **Start** and **Stop** buttons. There is a logical reason for it as while writing code, we can't move to the left outside the left margin. Left margin is the boundary of the screen and indentation is made towards the right side. So we follow the construct that is equivalent to the program being written. The major advantage of this approach is achieved when we draw a flowchart of solution of a complex problem. The flowchart is the logical depiction of the solution to the problem. One can write code easily with the help of the flowchart. There will be one to one correspondence between the segments of the flowcharts and the code.

Loops

Going on from the decision structures we discussed about loops. In our program if we have to do something repeatedly then we can think of applying loop structure there. There are few variants of loops in C language. However, other languages might have lesser number of loop variants but a programming language always has loops constructs.

vubookshoppk.com

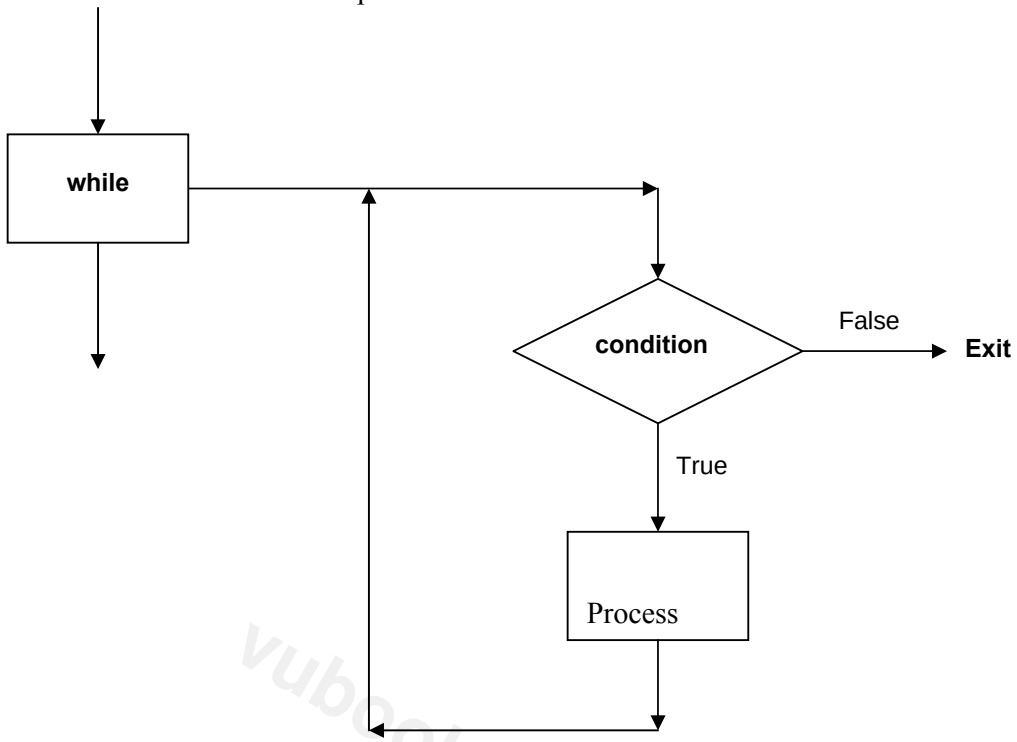
While Loop

The syntax of the while loop is as follows:

```
while ( condition )
{       // while code block

        // Do something

}
```

The condition is a logical expression like a == b that returns true or false. Braces are mandatory to for while loop when there are multiple lines of code inside the while code block. If there is only single line inside the while code block, the braces become optional. It is good practice to use braces. The statements inside the while code block are never executed, if the while condition results in false for very first time it is entered. In other words, statements inside the while code block executes 0 to n times.

The flowchart for the while loop is as follows:



vubookshoppk.com

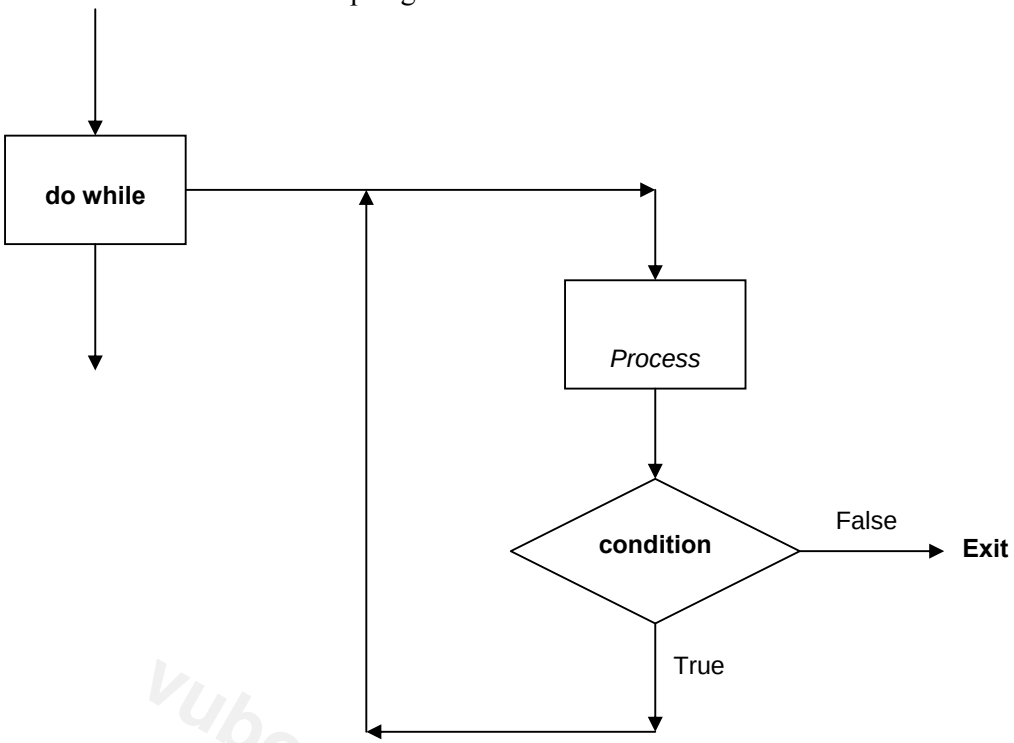
Do-While Loop

Next loop variant is Do-while. It syntax is as under

```
do
{       // do-while code block

        // Do something

}
while ( condition )
```

The important difference of this loop from the rest ones is that it is executed once before the condition is evaluated. That means the statements of do-while code block execute at least once.

The flowchart for do-while loop is given below:



vubookshoppk.com

For Loop

The **for loop** becomes bread and butter for us as it gathers three things together. The syntax for the for loop is as follows:

```
for ( initialization statements; condition; incremental statements)
{       //for code block

        // Do something

}
```

E.g.,

```
for ( int i = 0; i  <  10; i ++)
{

}
```

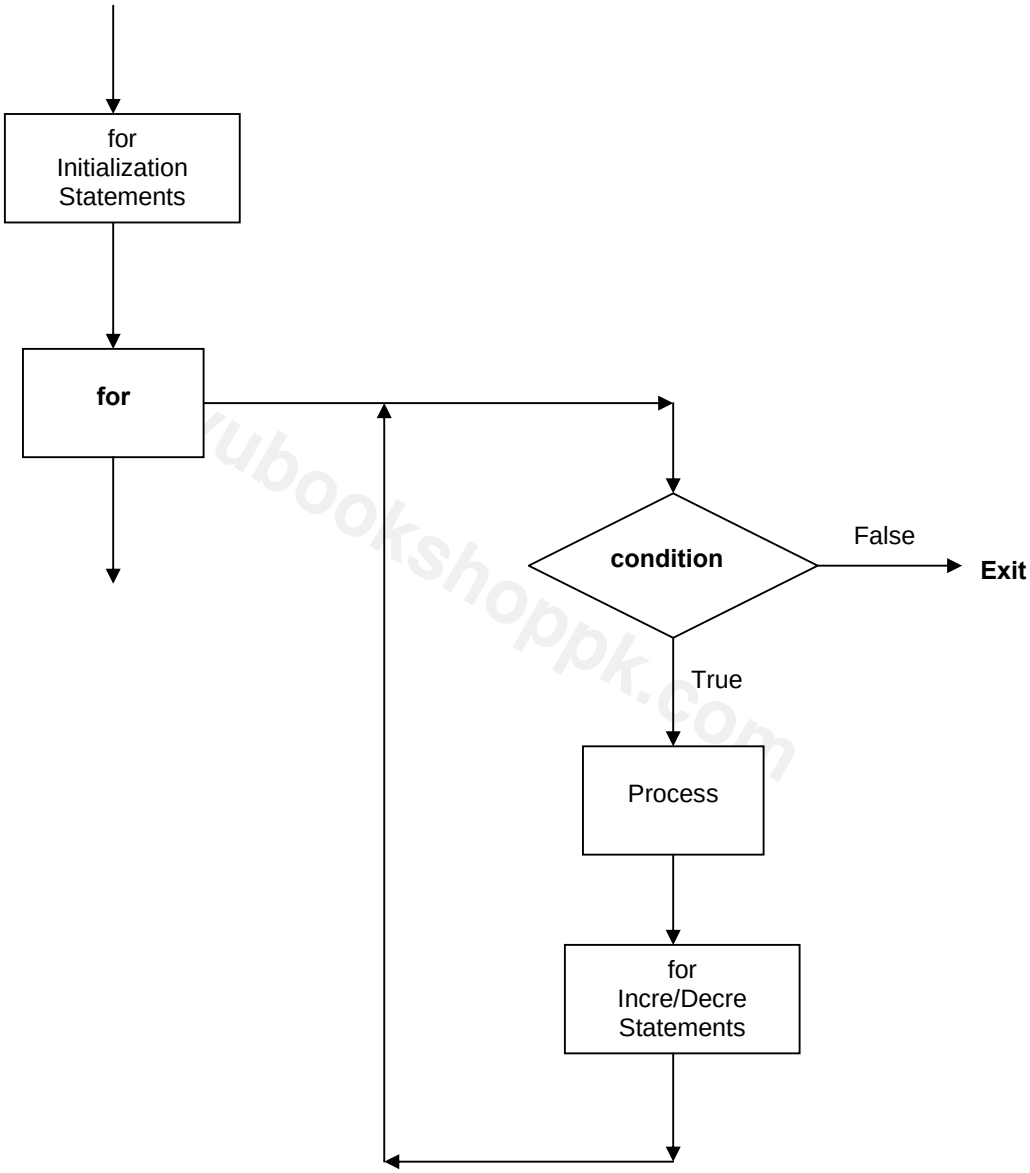
The **for loop** is executed until the condition returns true otherwise it is terminated. The braces are not mandatory if there is single statement in the for code block. But for sake of good programming practice, the single statement is also enclosed in braces. Some people write the for loop in the following manner:

```
for ( initialization statements; condition; incremental statements){
        //for code block
```

```
        // Do something
}
```

Both the methods for writing of for loop are perfectly correct. You can use anyone of these. If you indent your code properly, the process will become easier.

The flowchart for **for loop** is as under:



vubookshoppk.com

vubookshoppk.com

switch, break and continue Statements

For **multi-way decisions**, we can use nested if-statements or separate if-statements or **switch** statement. There are few limitations of switch statement but it is necessary to use **break** statements in every **case** inside the switch statement. If a **case** results in true when there is no **break** statement inside it, all the statements below this **case** statement are executed. **break** statement causes to jump out of the switch statement. We use **break** at the end of every **case** statement. By using **break**, the jumping out from **switch** statement is in a way bit different from the rules of structured programming. But **break** statement is so elegant and useful that you can use it inside **switch** statement and inside loops. If we use break inside a loop, it causes that loop to terminate. Similarly **continue** statement is very useful inside loops. **continue** statement is used, when at a certain stage, you don't want to execute the remaining statements inside your loop and want to go to the start of the loop.

Functions

In C/C++, functions are a way of modularizing the code. A bigger problem is broken down into smaller and more manageable parts. There is no rule of thumb for the length of each part but normally one function's length is not more than one screen.

Function Calling

We covered Functions Calling by value and by reference. The default of C language is call by value. Call by value means that when we call a function and pass some

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

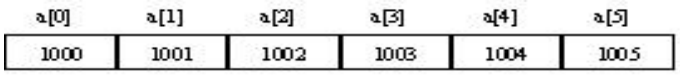
parameter to it, the calling function gets the copy of the value and the original value remains unchanged. On the other hand, sometimes, we want to call a function and want to see the changed value after the function call then call by reference mechanism is employed. We achieved call by reference by using Pointers. Remember while calling functions, call by value and call by reference are different techniques and default for ordinary variables is call by value.

Top-Down Methodology

We discussed top-down design methodology. How do we see a problem at a high level and identify major portions of it. Then by looking at each portion we identify smaller parts inside it to write them as functions.

Arrays

After discussing functions and playing little bit with function calling, we had elaborated the concept of Arrays. As discussed previously in this lecture, arrays are



used to aggregate variables of same data type. We wrote little functions about it and did some exercises e.g., when we wanted to store age of students of our class. Then instead of using a separate variable for each student, an array was employed to store the ages of the students. Then to manipulate or to access individual array elements, a technique **array indexing** was used. One important point to remember is that array indexes start from 0. Let's say our array name is **a** of 10 **int**s, its first element will be a[0] while the last one will be a[9]. Other languages like Fortran carry out 1-based indexing. Due to this 0 based indexing for arrays in C language, programmers prefer to start loops from 0.

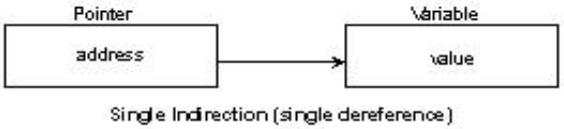
Arrays can also be multi-dimensional. In C language, arrays are stored in row major order that a row is stored at the end of the previous row. Because of this storage methodology, if we want to access the first element of the second row then we have to jump as many numbers as the number of columns in the first row. This fact becomes important when we are passing arrays to functions. In the receiving function parameters, we have to write all the dimensions of the array except the extreme-left one. When passing arrays to functions, it is always call by reference by default; it is not call by value as in the default behavior of ordinary variables. Therefore, if the called function changes something in the array, that change is actually made in the original array of the calling function. When we pass ordinary variables to functions, they are passed by value because of the default behavior. But when an array is passed to a function, the default behavior changes and array is passed by reference. We also did some examples of arrays by using Matrices and did some exercises by transposing and reversing a squared matrix. Arrays are not just used in Mathematics or Linear Algebra but are employed in a number of other problems like when we store ages, names, and grades or want to calculate grade point of average. This is very useful

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

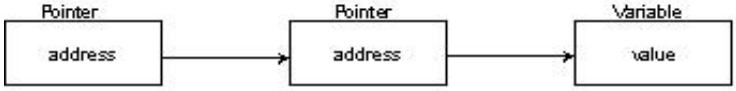
construct especially when used with loops. Normally it is very rare that you see an array in a program and loop is not being used to manipulate it.
Like nested if-statements, we have nested loops, used with multi-dimensional arrays. A while loop can have an inner while loop. Similarly a for loop can have a for loop inside. It is also not necessary that a **while loop** should have only a **while loop** but it can be a **for loop** also or any other construct like **if-statement**.

Pointers

It is very important topic of C/C++ . Pointers are different types of variables that contain memory address of a variable instead of a value.



Single Indirection (single dereference)



Double Indirection (double dereference)

The very first example we discussed for pointers was for implementing function calling by reference. Suppose we want to interchange (swap) two numbers by making a function call. If we pass two variables to the function, these will be passed as ordinary variables by value. Therefore, it will be ineffective as swapping of variables inside the function will only be on the copies and not on the original variables. So instead of passing variables we pass their addresses. In the called function, these addresses are taken into pointer variables and pointers start pointing the original variables. Therefore, the swapping operation done inside the function is actually carried out on the original variables.
We also saw that Pointers and Arrays are inter-linked. The array name itself is a pointer to the first element. It is a constant pointer that cannot be incremented like normal pointer variables. In case of two-dimensional arrays, it points to the first row and first column. In three-dimensional array, you can imagine it pointing to the front corner of the cube.

File I/O

We discussed about Files and File I/O for sequential and random files. We used a mixture of C/C++ for file handling and how the sequential and random files are accessed. We saw several modes of opening files. The important functions were **seek** and **tell** functions. Seek functions (seekg and seekp) used to move into the file and tell functions (tellg and tellp) provided us the location inside the file.

You are required to go with very clear head, try to understand concepts and assess how much you have learned so far to prepare for the mid-term examination.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Lecture No. 23

Reading Material

Deitel & Deitel - C++ How to Program Chapter. 17

Summary

- Pre-processor
- include directive
- define directive
- Other Preprocessor Directives
- Macros
- Example
- Tips

vubookshoppk.com

Preprocessor

Being a concise language, C needs something for its enhancement. So a preprocessor is used to enhance it. It comes with every C compiler. It makes some changes in the code before the compilation. The compiler gets the modified source code file. Normally we can't see what the preprocessor has included. We have so far been using *#include* preprocessor directive like *#include<iostream.h>*. What actually *#include* does? When we write *#include<somefile>*, this *somefile* is ordinary text file of C code. The line where we write the *#include* statement is replaced by the text of that file. We can't see that file included in our source code. However, when the compiler starts its work, it sees all the things in the file. Almost all of the preprocessor directives start with # sign. There are two ways to use *#include*. We have so far been including the file names enclosing the angle brackets i.e. *#include <somefile>*. This way of referring a file tells the compiler that this file exists in some particular folder (directory) and should be included from there. So we have included *iostream.h*, *stdlib.h*, *fstream.h*, *string.h* and some other files and used angle brackets for all of these files. These files are located in a specific directory. While using the *Dev-Cpp* compiler, you should have a look at the directory structure. Open the *Dev-Cpp* folder in the windows explorer, you will see many subfolders on the right side. One of these folders is '*include'*. On expansion of the folder '*include'*, you will see a lot of files in this directory. Usually the extension of these files is 'h'. Here 'h' stands for header files. Normally we add these files at the start of the program. Therefore these are known as header files. We can include files anywhere in the code but it needs to be logical and at the proper position.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

include directive

As you know, we have been using functions in the programs. If we have to refer a function (call a function) in our program, the prototype of function must be declared before its usage. The compiler should know the name of the function, the arguments it is expecting and the return type. The first parse of compilation will be successful. If we are using some library function, it will be included in our program at the time of linking. Library functions are available in the compiled form, which the linker links with our program. After the first parse of the compiler, it converts the source code into object code. Object code is machine code but is not re-locateable executable. The object code of our program is combined with the object code of the library functions, which the program is using. Later, some memory location information is included and we get the executable file. The linker performs this task while the compiler includes the name and arguments of the function in the object code. For checking the validity of the functions, the compiler needs to know the definition of the function or at least the prototype of the function. We have both the options for our functions. Define the function in the start of the program and use it in the main program. In this case, the definition of the function serves as both prototype and definition for the function. The compiler compiles the function and the main program. Then we can link and execute it. As the program gets big, it becomes difficult to write the definitions of all the functions at the beginning of the program. Sometimes, we write the functions in a different file and make the object file. We can include the prototypes of these functions in our program in different manners. One way is to write the prototype of all these functions in the start before writing the program. The better way is to make a header file (say myheaderfile.h) and write the prototypes of all the functions and save it as ordinary text file. Now we need to include it in our program using the *#include* directive. As this file is located at the place where our source code is located, it is not included in the angle brackets in *#include* directive. It is written in quotation marks as under:

```
#include “myHeaderFile.h”
```

The preprocessor will search for the file *“myHeaderFile.h”* in the current working directory. Let’s see the difference between the process of the including the file in brackets and quotation marks. When we include the file in angle brackets, the compiler looks in a specific directory. But it will look into the current working directory when the file is included in quotation marks. In the Dev-Cpp IDE, under the *tools* menu option, select *compiler options*. In this dialogue box, we can specify the directories for libraries and include files. When we use angle brackets with *#include*, the compiler will look in the directories specified in include directories option. If we want to write our own header file and save it in ‘My Document’ folder, the header file should be included with the quotation marks.

When we compile our source code, the compiler at first looks for the include directives and processes them one by one. If the first directive is *#include<iostream.h>*, the compiler will search this file in the include directory. Then it will include the complete header file in our source code at the same position where the ‘include directive’ is written. If the 2nd include directive contains another file, this file will also be included in the source code after the *iostream.h* and so on. The

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

compiler will get this expanded source code file for compilation. As this expanded source code is not available to us and we will get the executable file in the end.
Can we include the header file at the point other than start of the program? Yes. There is no restriction. We can include wherever we want. Normally we do this at the start of the program as these are header files. We do not write a portion of code in a different file and include this file somewhere in the code. This is legal but not a practice. We have so far discussed include directive. Now we will discuss another important directive i.e. define directive.

define directive

We can define macros with the *#define* directive. Macro is a special name, which is substituted in the code by its definition, and as a result, we get an expanded code. For example, we are writing a program, using the constant Pi. Pi is a universal constant and has a value of 3.1415926. We have to write this value 3.1415926 wherever needed in the program. It will be better to define Pi somewhere and use Pi instead of the actual value. We can do the same thing with the variable Pi as *double Pi = 3.1415926* while employing Pi as variable in the program. As this is a variable, one can re-assign it some new value. We want that wherever we write Pi, its natural value should be replaced. Be sure that the value of Pi can not be changed. With the define directive, we can define Pi as:

```
#define PI 3.1415926
```

We need to write the name of the symbolic constant and its value, separated by space. Normally, we write these symbolic constants in capitals as it can be easily identifiable in the code. When we request the compiler to compile this file, the preprocessor looks for the define directives and replaces all the names in the code, defined with the define directives by their values. So compiler does not see PI wherever we have used PI is replaced with 3.1415926 before the compiler compiles the file.

A small program showing the usage of #define.

```
/* Program to show the usage of define */

#include <iostream.h>

#define PI 3.1415926            // Defining PI

main()
{
   int radius = 5;
   cout << "Area of circle with radius " << radius << " = " << PI * radius * radius;
}
```

What is the benefit of using it? Suppose we have written a program and are using the value of PI as 3.14 i.e. up to two decimal places. After verifying the accuracy of the result, we need to have the value of PI as 3.1415926. In case of not using PI as define, we have to search 3.14 and replace it with 3.1415926 each and every place in the source code. There may be a problem in performing this 'search and replace' task. We

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

can miss some place or replace something else. Suppose at some place, 3.14 is representing something else like tax rate. We may change this value too accidentally, considering it the value for PI. So we can't conduct a blind search and replace and expect that it will work fine. It will be nicer to define PI at the start of the program. We will be using PI instead of its value i.e. 3.1415926. Now if we want to change the value of PI, it will be changed only at one place. The complete program will get the new value. When we define something with the *#define* directive, it is substituted with the value before the compiler compiles the file. This gives us a very nice control needed to change the value only at one place. Thus the complete program is updated.

We can also put this definition of PI in the header file. The benefit of doing this is, every program which is using the value of PI from this header file, will get the updated value when the value in header file is changed. For example, we have five functions, using the PI and these functions are defined in five different files. So we need to define PI (i.e. *#define PI 3.1415926*) in all the five source files. We can define it in one header file and include this header file in all the source code files. Each function is getting the value of PI from the header file by changing the value of PI in the header file, all the functions will be updated with this new value. As these preprocessor directives are not C statements, so we do not put semicolon in the end of the line. If we put the semicolon with the *#include* or *#define*, it will result in a syntax error.

vubookshoppk.com

Other Preprocessor Directives

There are some other preprocessor directives. Here is the list of preprocessor directives.

- #include <filename>
- #include "filename"
- #define
- #undef
- #ifdef
- #ifndef
- #if
- #else
- #elif
- #endif
- #error
- #line
- #pragma
- #assert

All the preprocessor directives start with the sharp sign (#). We can also do conditional compilation with it. We have *#if, #else, #endif* and for else if *#elif* is used. It can also be checked whether the symbol which we have defined with *#define,* is available or not. For this purpose, *#ifdef* is used. If we have defined PI, we can always say:

```
#ifdef PI
  … Then do something
```

```
#endif
```

This is an example of conditional compilation. If a symbolic constant is defined, it will be error to define it again. It is better to check whether it is already defined or not. If it is already defined and we want to give it some other value, it should be undefined first. The directive for undefine is *#undef*. At first, we will undefine it and define it again with new value. Another advantage of conditional compilation is 'while debugging'. The common technique is to put output statements at various points in the program. These statements are used in the code to check the value of different variables and to verify that the program is working fine. It is extremely tedious to remove all these output statements which we have written for the debugging. To overcome this problem, we can go for conditional compilation. We can define a symbol at the start of the program as:

```
#define DEBUG
```

Here we have defined a symbol *DEBUG* with no value in front of it. The value is optional with the define directive. The output statements for debugging will be written as:

```
#ifdef DEBUG
        cout << "Control is in the while loop of calculating average";
#endif
```

Now this statement will execute if the *DEBUG* symbol is defined. Otherwise, it will not be executed.

Here is an example using the debug output statements:

```
// Program that shows the use of Define for debugging
// Comment the #define DEBUG and see the change in the output

#include <iostream.h>
#include <stdlib.h>

#define DEBUG

main()
{
        int z ;
        int arraySize = 100;
        int a[100] ;
        int i;

        // Initializing the array.
        for ( i = 0; i < arraySize; i++ )
        {
                a[i] = i;
        }

        // If the symbol DEBUG is defined then this code will execute
```

vubookshoppk.com

```
        #ifdef DEBUG
                for ( i = 0 ; i < arraySize ; i ++ )
                    cout << "\t " << a[i];
    #endif

        cout << " Please enter a positive integer " ;
        cin >> z ;
        int found = 0 ;

        // loop to search the number.
        for ( i = 0 ; i < arraySize ; i ++ )
        {
                if ( z == a[i] )
                {
                        found = 1 ;
                        break ;
                }
        }
        if ( found == 1 )
                cout << " We found the integer at position " << i ;
        else
                cout << " The number was not found " ;
}
```

With preprocessor directives, we can carry out conditional compilation, a macro translation that is replacement of a symbol by the value in front of it. We can not redefine a symbol without undefining it first. For undefining a symbol, *#undef* is used. e.g. the symbol PI can be undefined as:

```
#undef PI
```

Now from this point onward in the program, the symbol PI will not be available. The compiler will not be able to view this symbol and give error if we have used it in the program after undefining.

As an exercise, open some header files and read them. e.g. we have used a header file *conio.h* (i.e. *#define<conio.h>*) for consol input output in our programs. This is legacy library for non-graphical systems. We have two variants of *conio* in Dev-Cpp i.e. *conio.h* and *conio.c* (folder is 'Dev-Cpp\include'). Open and read it. Do not try to change anything, as it may cause some problems. Now you have enough knowledge to read it line by line. You will see different symbols in it starting with underscore (_). There are lots of internal constants and symbolic names starting with double underscore. Therefore we should not use such variable names that are starting with underscore. You can find the declaration of different functions in it e.g. the function *getche()* (i.e. get character with echo) is declared in *conio.h* file. If we try to use the function *getche()* without including the *conio.h* file, the compiler will give error like 'the function *getche()* undeclared'. There is another interesting construct in *conio.h* i.e.

```
#ifdef __cplusplus
```

vubookshoppk.com

```
        extern "C" {
#endif
```

If the symbol *__cplusplus* is defined, the statement '*extern "C" {* ' will be included in the code. We have an opening brace here. Look where the closing brace is. Go to the end of the same file. You will find the following:

```
#ifdef __cplusplus
        }
#endif
```

This is an example of conditional compilation i.e. if the symbol is defined, it includes these lines in the code before compiling. Go through all the header files, we have been using in our programs so that you can see how professional programmers write code. If you have the linux operating system, it is free with a source code. The source code of linux is written in C language. You can see the functions written by the C programming Gurus. There may be the code of string manipulation function like string copy, string compare etc.

Macros

Macros are classified into two categories. The first type of macros can be written using *#define.* The value of PI can be defined as:

```
#define PI 3.1415926
```

Here the symbol PI will be replaced with the actual value (i.e. 3.1415926) in the program. These are simple macros like symbolic names mapped to constants.

In contrast, the second type of macros takes arguments. It is also called a parameterized macros. Consider the following:

```
#define square(x) x * x
```

Being a non-C code, it does not require any semicolon at the end. Before the compiler gets the file, the macro replaces all the occurrences of square (x) (that may be square (i), square (3) etc) with (x * x) (that is for square (i) is replaced by i * i, square(3) is replaced by 3 * 3). The compiler will not see square(x). Rather, it will see x * x, and make an executable file. There is a problem with this macro definition as seen in the following statement.

```
square (i + j);
```

Here we have *i+j* as *x* in the definition of macro. When this is replaced with the macro definition, we will get the statement as:

```
i + j * i + j
```

This is certainly not the square of *i + j.* It is evaluated as *(i + (j * i) + j* due to the precedence of the operators. How can we overcome this problem? Whenever you

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

write a parameterized macro, it is necessary to put the parenthesis in the definition of macro. At first, write the complete definition in the parenthesis, and then put the *x* also in parenthesis. The correct definition of the macro will be as:

```
#define square(x) ((x) * (x))
```

This macro will work fine. When this macro definition is replaced in the code, parenthesis will also be copied making the computation correct.

Here is a sample program showing the use of a simple square macro:

```
/* Program to show the use of macro */

#include <iostream.h>

// Definition of macro square
#define square(x) ((x) * (x))

main()
{
   int x;

   cout << endl;
   cout << " Please enter the value of x to calculate its square ";
   cin >> x;
   cout << " Square of x = " << square(x) << endl;
   cout << " Square of x+2 = " << square(x+2) << endl;
   cout << " Square of 7 = " << square(7);
}
```

vubookshoppk.com

We can also write a function to *square(x)* to calculate the square of a number. What is the difference between using this *square(x)* macro and the *square(x)* function? Whenever we call a function, a lot of work has to be done during the execution of the program. The memory in machine is used as stack for the program. The state of a program (i.e. the value of all the variables of the program), the line no which is currently executing etc is on the stack. Before calling the function, we write the arguments on the stack. In a way, we stop at the function calling point and the code jumps to the function definition code. The function picks up the values of arguments from the stack. Do some computation and return the control to the main program which starts executing next line. So there is lot of overhead in function calling. Whenever we call a function, there is some work that needed to be done. Whenever we do a function call, like if we are calling a function in a loop, this overhead is involved with every iteration. The overhead is equal number of times the loop executed. So computer time and resources are wasted. Obviously there are a number of times when we need to call functions but in this simple example of calculating square, if we use square function and the program is calling this function 1000 times, a considerable time is wasted. On the other hand, if we define square macro and use it. The code written in front of macro name is substituted at all the places in the code where we are using square macro. Therefore the code is expanded before compilation and compiler see ordinary multiplication statements. There is no function call

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

involved, thus making the program run faster. We can write complex parameterized macros. The advantage of using macros is that there is no overhead of function calls and the program runs faster. If we are using lot of macros in our program, it is replaced by the macro definition at every place in the code making the program bloat. Therefore our source code file becomes a large file, resulting in the enlargement of the executable file too. Sometimes it is better to write functions and define things in it. For simple things like taking a square, it is nice to write macros that are only one line code substitution by the preprocessor.

Take care of few things while defining macros. There is no space between the macro name and the starting parenthesis. If we put a space there, it will be considered as simple macro without parameters. We can use more than one argument in the macros using comma-separated list. The naming convention of the arguments follows the same rules as used in case of simple variable name. After writing the arguments, enclosing parenthesis is used. There is always a space before starting the definition of the macro.

Example

Suppose we have a program, which is using the area of circle many times in it. Therefore we will write a macro for the calculation of the area of circle. We know that the formula for area of circle is $PI*r^2$. Now this formula is substituted wherever we will be referring to this macro. We know that the PI is also a natural constant. So we will define it first. Then we will define the macro for the area of the circle. From the perspective of visibility, it is good to write the name of the macro in capital as CIRCLEAREA. We don't need to pass the PI as argument to it. The only thing, needed to be passed as argument, is radius. So the name of the macro will be as CIRCLEAREA (X).We will write the formula for the calculation of the area of the circle as:

```
#define CIRCLEAREA(X) (PI * (X) * (X))
```

Here is the complete code of the program:

```
/* A simple program using the area of circle formula as macro */

#include <iostream.h>

// Defining the macros
#define PI 3.14159
#define CIRCLEAREA(X) ( PI * X * X)

main()
{
        float radius;
        cout << " Enter radius of the circle: ";
        cin >> radius;
        cout << " Area of circle is " << CIRCLEAREA (radius);
}
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

The CIRCLEAREA will be replaced by the actual macro definition including the entire parenthesis in the code before compilation. As we have used the parenthesis in the definition of the CIRCLEAREA macro. The statement for ascertaining the area of circle with double radius will be as under:

```
CIRCLEAREA(2 * radius);
```

The above statement will work fine in calculating the correct area. As we are using multiplication, so it may work without the use of parenthesis. But if there is some addition or subtraction like CIRCLEAREA(radius + 2) and the macro definition does not contain the parenthesis, the correct area will not be calculated. Therefore always use the parenthesis while writing the macros that takes arguments.

There are some other things about header files. As a proficient programmer writing your own operating systems, you will be using these things. There are many operating systems, which are currently in use. Windows is a popular operating system, DOS is another operating system for PC's, Linux, and different variety of Unix, Sun Solaris and main frame operating systems. The majority of these operating systems have a C compiler available. C is a very elegant operating systems language. It is very popular and available on every platform. By and large the source code which we write in our programs does not change from machine to machine. The things, which are changed, are system header files. These files belong to the machine. The header files, which we have written for our program, will be with the source code. But the *iostream, stdlib, stdio, string* header files have certain variations from machine to machine. Over the years as the C language has evolved, the names of these header files have become standard. Some of you may have been using some other compiler. But you have noted that in those compilers, the header files are same, as *iostream.h, conio.h* etc are available. It applies to operating systems. While changing operating systems, we come up with the local version of C/C++ compiler. The name of the header files remains same. Therefore, if we port our code from one operating system to another, there is no need to change anything in it. It will automatically include the header files of that compiler. Compile it and run it. It will run up to 99 % without any error. There may be some behavioral change like function *getche()* sometimes read a character without the enter and sometimes you have to type the character and press enter. So there may be such behavioral change from one operating system to other. Nonetheless these header files lead to a lot of portability. You can write program at one operating system and need not to take the system header file with the code to the operating system.

On the other hand, the header files of our program also assist in the portability in the sense that we have all the function prototypes, symbolic definitions, conditional compilations and macros at one place. While writing a lot of codes, we start writing header files for ourselves because of the style in which we work. We have defined some common functions in our header files. Now when we are changing the operating system, this header file is ported with the source code. Similarly, on staring some program, we include this header file because it contains utility function which we have written.

Here is an interesting example with the *#define*. If you think you are sharp here is a challenge for you. Define you own vocabulary with the #define and write C code in

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

front of it. One can write a poem using this vocabulary which will be replaced by the preprocessor with the C code. What we need is to include one header file that contains this vocabulary. So an ordinary English poem is actually a C code. Interesting things can be done using these techniques.

Tips

- All the preprocessor directives start with the # sign
- A symbol can not be redefined without undefining it first
- The conditional compilation directives help in debugging the program
- Do not declare variable names starting with underscore
- Always use parenthesis while defining macros that takes arguments

Lecture No. 24

vubookshoppk.com

Reading Material

Deitel & Deitel - C++ How to Program Chapter 15, 18

15.3, 18.10

Summary

1) Memory Allocation
2) Dynamic Memory Allocation
3) calloc Function
4) malloc Function
5) free ()
6) realloc Function
7) Memory Leak
8) Dangling Pointers
9) Examples
10) Exercise
11) Tips

Memory Allocation

After having a thorough discussion on static memory allocation in the previous lectures, we will now talk about dynamic memory allocation. In this lecture, the topics being dilated upon include- advantages and disadvantages of these both types of memory allocation and the common errors, which usually take place while programming with dynamic memory allocation. Let's first talk about the dynamic memory allocation.

Dynamic Memory Allocation

Earlier, whenever we declared arrays, the size of the arrays was predefined. For example we declared an array of size 100 to store ages of students. Besides, we need 20, 25 or 50 number of students to store their ages. The compiler reserves the memory to store 100 integers (ages). If there are 50 integers to be stored, the memory for remaining 50 integers (that has been reserved) remains useless. This was not an important matter when the programs were of small sizes. But now when the programs grow larger and use more resources of the system, it has become necessary to manage the memory in a better way. The dynamic memory allocation method can be helpful in the optimal utilization of the system.

It is better to compare both the static and dynamic allocation methods to understand the benefits of the usage of dynamic memory allocation. In static memory, when we write the things like int i, j, k ; these reserve a space for three integers in memory. Similarly the typing of char s[20] will result in the allocation of space for 20 characters in the memory. This type of memory allocation is static allocation. It is also known as compile time allocation. This memory allocation is defined at the time when we write the program while exacting knowing how much memory is required.

Whenever, we do not know in advance how much memory space would be required, it is better to use dynamic memory allocation. For example if we want to calculate the average age of students of a class. Instead of declaring an array of large number to allocate static memory, we can ask number of students in the class and can allocate memory dynamically for that number. The C language provides different functions to allocate the memory dynamically.

The programs, in which we allocate static memory, run essentially on stack. There is another part of memory, called heap. The dynamic memory allocation uses memory from the heap. All the programs executing on the computer are taking memory from it for their use according to the requirement. Thus heap is constantly changing in size. Windows system may itself use memory from this heap to run its processes like word processor etc. So this much memory has been allocated from heap and the remaining is available for our programs. The program that will allocate the memory dynamically, will allocate it from the heap.

Let's have a look on the functions that can be used to allocate memory from the heap.

Before actually allocating memory, it is necessary to understand few concepts. We have already studied these concepts in the lectures on 'pointers'. Whenever we allocate a memory what will we get? We need to be careful about that. When we say **int i,** a space is reserved for an integer and it is labeled as **i**. Here in dynamic memory, the situation is that the memory will be allocated during the execution of the program. It is difficult to determine whether the memory allocated is an array, an integer, 20 integers or how much space is it? To over this uncertainty, we have to use pointers.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Whenever we allocate any memory from the heap, the starting position of the block of the memory allocated is returned as an address that is kept in a pointer. Then we manipulate the memory with the help of this pointer. We have to introduce a new type of a pointer, called '**void'**. We have used the pointers of type- int, char, float etc. For these, we write like **int *i ;** which means **i** is a pointer to an integer. In this case, the compiler automatically knows that **i** has the address of the memory, occupied by an integer. Same thing applies when we write **char *s .** It means **s** is a pointer to a character data type. So every pointer we have used so far pointed to a specific data type.

The functions used for dynamic memory allocation, provide a chunk of memory from heap. The function does not know for what data type this chunk of memory will be used? It returns a pointer of type void. A pointer ptr of type void is declared as under.

void *ptr ;

The 'void' is a special type of pointers. We have to cast it before its use. The cast means the conversion of 'void' into a type of pointer that can be used for native data type like int, char, float etc. The operator used for casting, in C, is standard cast operator. We write the name of the type in parentheses. Suppose we have a pointer ptr defined as a void pointer like

void *ptr ;

Before using this pointer to point to a set of integers, we will at first cast it. It means that it will be converted into a type of a pointer to an integer. The syntax of doing this casting is simple and is given below.

(int *) ptr ;

Here both int and * are written in parentheses. The **int** is the data type into which we are converting a **void** pointer **ptr**. Now **ptr** is a pointer to an integer. Similarly, we can write char, float and double instead of 'int', to convert **ptr** into a pointer to char, float and double respectively.

Casting is very useful in dynamic memory allocation. The memory allocation functions return a chunk of memory with a pointer of type void. While storing some type of data , we at first, cast the pointer to that type of data before its usage. It is an error to try to use the void pointer and dereference it. In case, we write *ptr and use it in an expression, there will be an error. So we have to cast a void pointer before its use.

Another interesting aspect of pointer is the NULL value. Whenever we define a pointer or declare a pointer, normally, it is initialized to a NULL value. NULL has been defined in the header files stdlib.h and stddef.h. So at least one of these files must be included in the program's header to use the NULL. A NULL pointer is a special type of pointer with all zeros value. All zeros is an invalid memory address. We can't use it to store data or to read data from it. It is a good way to ascertain whether a pointer is pointing to a valid address or has a NULL value.

calloc Function

The syntax of the calloc function is as follows.

void *calloc (size_t n, size_t el_size)

This function takes two arguments. The first argument is the required space in terms of numbers while the second one is the size of the space. So we can say that we require **n** elements of type **int**. We have read a function **sizeof.** This is useful in the cases where we want to write a code that is independent of the particular machines that we are running on. So if we write like

void calloc(1000, sizeof(int))

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

It will return a memory chunk from the heap of 1000 integers. By using **sizeof (int)** we are not concerned with the size of the integer on our machine whether it is of 4 bytes or 8 bytes. We will get automatically a chunk that can hold 1000 integers. The said memory will be returned if a chunk of similar size is available on the heap. Secondly, this memory should be available on heap in continuous space. It should not be in split blocks. The function returns a pointer to the starting point of the allocated memory. It means that if starting point of the chunk is gotten, then the remaining memory is available in a sequence from end to end. There cannot be gaps and holes between them. It should be a single block. Now we have to see what happens when either we ask for too much memory at a time of non-availability of enough memory on the heap or we ask for memory that is available on the heap , but not available as a single chunk?. In this case, the call to calloc will fail. When a call to memory allocation functions fails, it returns a NULL pointer. It is important to understand that whenever we call a memory allocation function, it is necessary to check whether the value of the pointer returned by the function is NULL or not. If it is not NULL, we have the said memory. If it is NULL, it will mean that either we have asked for too much memory or a single chunk of that size is not available on the heap.
Suppose, we want to use the memory got through calloc function as an integer block We have to cast it before using. It will be written as the following statement.

(int *) calloc (1000, sizeof (int)) ;

Another advantage of calloc is that whenever we allocate memory by using it. The memory is automatically initialized to zeros. In other words it is set to zeros. For casting we normally declare a pointer of type which we are going to use. For example, if we are going to use the memory for integers. We declare an integer pointer like **int *iptr;** Then when we allocate memory through calloc, we write it as

iptr = (int *) calloc (1000, sizeof(int)) ;

(int *) means cast the pointer returned by calloc to an integer pointer and we hold it in the declared integer pointer **iptr**. Now **iptr** is a pointer to an integer that can be used to manipulate all the integers in that memory space. You should keep in mind that after the above statement, a NULL check of memory allocation is necessary. An 'if statement' can be used to check the success of the memory allocation. It can be written as under

```
if (iptr == NULL)
        any error message or code to handle error ;
```

If a NULL is returned by the calloc, it should be treated according to the logic so that the program can exit safely and it should not be crashed.

The next function used for allocating memory is malloc.

malloc Function

The malloc function takes one argument i.e. the number of bytes to be allocated. The syntax of the function is

void * malloc (size_t size) ;

It returns a void pointer to the starting of the chunk of the memory allocated from the heap in case of the availability of that memory. If the memory is not available or is fragmented (not in a sequence), malloc will return a NULL pointer. While using malloc, we normally make use sizeof operator and a call to malloc function is written in the following way.

malloc (1000 * sizeof(int)) ;

Here * is multiplication operator and not a dereference operator of a pointer.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

In the above call, we request for 1000 spaces in the memory each of the size, which can accommodate an integer. The 'sizeof(int)' means the number of bytes, occupied by an integer in the memory. Thus the above statement will allocate memory in bytes for 1000 integers. If on our machine, an integer occupies 4 bytes. A 1000 * 4 (4000) bytes of memory will be allocated. Similarly if we want memory for 1000 characters or 1000 floats, the malloc function will be written as

malloc (1000 * sizeof(char)) ;

and malloc (1000 * sizeof(float)) ;

respectively for characters and floats.

So in general, the syntax of malloc will be.

malloc (n * sizeof (*datatype*)) ;

where 'n' represents the numbers of required data type. The **malloc** function differs from **calloc** in the way that the space allocated by **malloc** is not initialized and contains any values initially.

Let's say we have a problem that states 'Calculate the average age of the students in your class.' The program prompts the user to enter the number of students in the class and also allows the user to enter the ages of the students. Afterwards, it calculates the average age. Now in the program, we will use dynamic memory. At first, we will ask the user 'How many students are in the class? The user enters the number of students. Let's suppose, the number is 35. This number is stored in a variable say '**numStuds'**. We will get the age of students in whole numbers so the data type to store age will be **int**. Now we require a memory space where we can store a number of integers equal to the value stored in **numStuds**. We will use a pointer to a memory area instead of an array. So we declare a pointer to an integer. Suppose we call it **iptr**. Now we make a call to calloc or malloc function. Both of them are valid. So we write the following statement

iptr = (int *) malloc (numStuds * sizeof (int)) ;

Now we immediately check **iptr** whether it has NULL value. If the value of **iptr** is not NULL, it will mean that we have allocated the memory successfully. Now we write a loop to get the ages of the students and store these to the memory, got through malloc function. We write these values of ages to the memory by using the pointer **iptr** with pointer arithmetic. A second pointer say **sptr** can be used for pointer arithmetic so that the original pointer **iptr** should remain pointing to the starting position of the memory. Now simply by incrementing the pointer **sptr**, we get the ages of students and store them in the memory. Later, we perform other calculations and display the average age on the screen. The advantage of this (using malloc) is that there is no memory wastage as there is no need of declaring an array of 50 or 100 students first and keep the ages of 30 or 35 students in that array. By using dynamic memory, we accurately use the memory that is required.

free ()

Whenever we get a benefit, there is always a cost. The dynamic memory allocation has also a cost. Here the cost is incurred in terms of memory management. The programmer itself has to manage the memory. It is the programmer's responsibility that when the memory allocated is no longer in use, it should be freed to make it a part of heap again. This will help make it available for the other programs. As long as the memory is allocated for a program, it is not available to other programs for use. So it is programmer's responsibility to free the memory when the program has done with it. To ensure it, we use a function **free**. This function returns the allocated memory,

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

got through **calloc** or **malloc,** back to the heap. The argument that is passed to this function is the pointer through which we have allocated the memory earlier. In our program, we write

free (iptr) ;

By this function, we call the memory allocated by **malloc** and pointed by the pointer **iptr** is freed. It goes back to the heap and becomes available for use by other programs. It is very important to note that whenever we allocate memory from the heap by using **calloc** or **malloc,** it is our responsibility to free the memory when we have done with it.

Following is the code of the program discussed above.

```
//This program calculates the average age of a class of students
//using dynamic memory allocation

#include <iostream.h>
#include <stdlib.h>
#include <string.h>

int main( )
{

        int numStuds, i, totalAge, *iptr, *sptr;
        cout <<"How many students are in the class ?    " ;
        cin >> numStuds;
        // get the starting address of the allocated memory in pointer iptr
        iptr = (int *) malloc(numStuds * sizeof(int));
        //check for the success of memory allocation
        if (iptr == NULL)
        {
                cout << "Unable to allocat space for " << numStuds << " students\n";
        return 1;
        // A nonzero return is usually used to indicate an error
        }
        sptr = iptr ; //sptr will be used for pointer arithmetic/manipulation
        i = 1 ;
        totalAge = 0 ;
        //use a loop to get the ages of students
        for (i = 1 ; i <= numStuds ; i ++)
        {
                cout << "Enter the age of student  " << i << " = " ;
        cin >> *sptr ;
        totalAge = totalAge + *sptr ;
        sptr ++ ;
        }
        cout << "The average age of the class is " << totalAge / numStuds << endl;
        //now free the allocated memory, that was pointed by iptr
        free (iptr) ;
        sptr = NULL ;
 }
```

vubookshoppk.com

Following is a sample out put of the program.

```
How many students are in the class ?  3
Enter the age of student  1 = 12
Enter the age of student  2 = 13
Enter the age of student  3 = 14
The average age of the class is 13
```

realloc Function

Sometimes, we have allocated a memory space for our use by **malloc** function. But we see later that some additional memory is required. For example, in the previous example, where (for example) after allocating a memory for 35 students, we wanted to add one more student. So we need same type of memory to store the new entry. Now the question arises 'Is there a way to increase the size of already allocated memory chunk ? Can the same chunk be increased or not? The answer is yes. In such situations, we can reallocate the same memory with a new size according to our requirement. The function that reallocates the memory is **realloc**. The syntax of **realloc** is given below.

void realloc (void * ptr, size_t size) ;

This function enlarges the space allocated to ptr (in some previous call of calloc or malloc) to a (new) size in bytes. This function receives two arguments. First is the pointer that is pointing to the original memory allocated already by using calloc or malloc. The second is the size of the memory which is a new size other than the previous size. Suppose we have allocated a memory for 20 integers by the following call of malloc and a pointer iptr points to the allocated memory.

(iptr *) malloc (20 * sizeof(int)) ;

Now we want to reallocate the memory so that we can store 25 integers. We can reallocate the same memory by the following call of realloc.

realloc (iptr, 25 * sizeof(int)) ;

There are two scenarios to ascertain the success of 'realloc'. The first is that it extends the current location if possible. It is possible only if there is a memory space available contiguous to the previously allocated memory. In this way the value of the pointer iptr is the same that means it is pointing to the same starting position, but now the memory is more than the previous one. The second way is that if such contiguous memory is not available in the current location, realloc goes back to the heap and looks for a contiguous block of memory for the requested size. Thus it will allocate a new memory and copy the contents of the previous memory in this new allocated memory. Moreover it will set the value of the pointer iptr to the starting position of this memory. Thus iptr is now pointing to a new memory location. The original memory is returned to the heap. In a way, we are handling dynamic arrays. The size of the array can be increased during the execution. There is another side of the picture. It may happen that we have stored the original value of iptr in some other pointer say sptr. Afterwards, we are manipulating the data through both the pointers. Then ,we use realloc for the pointer iptr. The realloc does not find contiguous memory with the original and allocates a new block of memory and points it by the pointer iptr. The original memory no longer exists now. The pointer iptr is valid now as it is pointing to the starting position of the new memory. But the other pointer sptr is no longer valid. It is pointing to an invalid memory that has been freed and may be is being used some other program. If we manipulate this pointer, very strange things can

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

happen. The program may crash or the computer may halt. We don't know what can happen. Now it becomes the programmer's responsibility again to make it sure that after realloc, the pointer(s) that have the value of the original pointer have been updated. It is also important to check the pointer returned by realloc for NULL value. If realloc fails, that means that it cannot allocate the memory. In this case, it returns a NULL value. After checking NULL value, (if realloc is successful), we should update the pointer that was referencing the same area of the memory.

We have noticed while getting powers of dynamic memory allocation, we face some dangerous things along with it. These are real problems. Now we will talk about the common errors that can happen with the memory allocation.

Memory Leak

The first problem may be the unreferenced memory. To understand this phenomenon, suppose, we allocate memory from heap and there is a pointer pointing to this memory. However, it is found that this pointer does not exist any more in our program. What will happen to the memory we had allocated. That chunk of memory is now unreferenced. Nothing is pointing to that memory. As there is no pointer to this memory, our program can't use it. Moreover, no other program can use it. Thus, this memory goes waste. In other words, the heap size is decreased as we had allocated memory from it despite the fact that it was never utilized. If this step of allocating memory and then destroy the pointer to this memory carries on then the size of the heap will going on to decrease. It may become of zero size. When there is no memory on heap, the computer will stop running and there may be a system crash. This situation is called a memory leak. The problem with memory leak is that you may be unaware of the memory leak caused by the program. Suppose there is 128 MB memory available on heap. We run our program that allocates 64 KB memory and terminates without freeing this memory. It does not effect but when if the memory is being allocated in a loop, that, suppose runs 1000 times and in each loop it allocates 64 KB of memory with out freeing the previous one. Then this program will try to allocate 64 * 1000 KB memory and at a certain point there will be no memory available and the program will crash. The same thing (no memory available) happens to other programs and the whole system locks up. So memory leak is a very serious issue.

This bug of memory leak was very common in the operating systems. This was a common thing, that the system was running well and fine for 4-5 hours and then it halted suddenly. Then the user had to reboot the system. When we reboot a system all the memory is refreshed and is available on the heap. People could not understand what was happening. Then there come the very sophisticated debugging techniques by which this was found that memory is being allocated continuously without freeing and thus the heap size becomes to zero. Thus memory is leaking out and it is no longer useable.

Let us see how does this happen and what we can do to prevent it. A simple way in which memory leak can happen is that suppose our main program calls a function. There, in the function, a pointer iptr is declared as a pointer to an integer. Then we call calloc or malloc in the function and allocate some memory. We use this memory and goes back to the main function without freeing this memory. Now as the pointer iptr has the function scope it is destroyed when the function exits. It is no longer there but the memory allocated remains allocated and is not being referenced as the pointer

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

pointing to it no longer exists. Now this memory is unreferenced which means it is leaked. This is a memory leak. Now if this function is being called repeatedly it means a chunk of memory is being allocated and is left unreferenced each time. Thus, each time a memory chunk from heap will be allocated and will become useless(as this will be unreferenced) and the heap size may become zero. As a programmer, it is our responsibility and a good rule of thumb will be that in which function the memory is allocated, it should be freed in the same function.

Sometimes the logic of the program is that the memory is being allocated somewhere and is being used somewhere else. It means we allocate memory in a function and use it in another function. In such situations, we should keep in mind that this all scenario is memory management and we have to take care of it. We allocate memory in a function and cannot free it here because it is being used in some other function. So we should have a sophisticated programming to make it sure that whenever we allocate a memory it should be freed somewhere or the other. Now it is not to do just with function calls. It also has to do when the program ends. Let consider, our program is running and we allocate memory somewhere and somewhere else there is a condition on which the program exits. If we exit without freeing the memory then there is a memory leak. The memory leakage is at operating system level. The operating system does not know that this memory is not being used by anyone now. From its aspect, some program is using this memory. So whenever we write program we should free the allocated memory wherever it is allocated. But at the program exit points we should do some task. This task is make it sure that when we allocated memory in the program this memory should be freed at exit points. The second necessary thing is that after freeing the memory, explicitly assign NULL to the pointer. Its benefit is that this pointer can be checked if it is pointing to some memory.

Whereas we do get this considerable flexibility in doing dynamic memory management, it is also our responsibility for freeing all the memory that we allocated from the heap. The other side of the coin is also that if we are using dynamic memory allocation in our program then we should check immediately if we have got memory. If we did not get (allocated) memory then exit the program in a good and safe way rather than to crash the program.

Dangling Pointers

Memory leak is one subtle type of error that can happen. There is another one. This other one is even more dangerous. This is dangling pointer. It has the inverse effect of the memory leak. Suppose, there was a pointer that was pointing to a chunk of memory, now by some reason that memory has deallocated and has gone back to heap. The pointer still has the starting address of that chunk. Now what will happen if we try to write something in the memory using this pointer? Some very strange thing can happen. This can happen that when we have put that memory back to heap some other program starts to use that memory. Operating system itself might have started using that memory. Now our program, by using that pointer try to write something in the memory that is being used by some other program. This may halt the machine as the position that is being tried to written may be a critical memory position. How does this situation arise? Lets consider a case. We have two pointers ptr1 and ptr2. These are pointers to integers. We allocate some memory from the heap by using calloc or malloc. The pointer ptr1 is pointing to the starting point of this allocated memory. To use this memory through a variable pointer we use the pointer ptr2. At start, we put the address of ptr1 in ptr2 and then do our processing with the help of ptr2. In the meantime, we go to exit the function. To free the allocated memory we use the pointer

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

ptr1. Thus the memory allocated goes back to heap and some other program may use it. The pointer ptr2 has the address of the same memory that it got from the ptr1. Now ptr2 points in a way to the memory that no longer belongs to the program. It has gone back to the heap. We can read the data residing at that memory location. But now if we try to write something in that location everything might break loose. We have to be very careful. The pointer ptr2 points to no location it is called dangling pointer. We have to be very careful about memory leak and dangling pointer.

The dynamic memory allocation is a very useful technique. In it what memory we require we take from the heap and use it and when it is no longer required we send it back to the heap. All the programs running on our machine (which are running on modern operating systems which are multitasking) work efficiently. They take memory of their requirement from the memory resources and return it back after using.

The sharing is not limited to memory resources this also include printers attached with the computer. The printer resource is being used by different programs like MS WORD, EXCEL and even may be by our program if we want to print something. We are also sharing the other resources like keyboard, monitor, and hard disk etc. But in terms of dynamic usage we are also sharing the memory. Our program in a way has to be a good neighbor to use the memory. It should use memory as long as it required and then after use it should give back this memory to the heap so that other programs can use this resource. So remember to free the memory it is as important as the allocation of memory.

So what interesting things we can do with memory allocation. A common thing in file handling is to copy a file. Our hard disks being electro mechanical devices are very slow. It is very expensive to access them. So while reading from them or writing to them we try that a big chunk should be written or read from them so that fewest disk writes and disk reads should occur. In order to do that, think combining dynamic memory allocation with disk read and write. Suppose we have to copy a file. We can easily find out the size of the file in bytes. Now we allocate this number of bytes from heap. If this size of memory is successfully allocated, we can say for a single file read of this allocated size. This means the entire file will be read to memory. This way we read a whole file with one command. Similarly, we can use a command to write the whole file. In this way we can be assured that we are doing the more efficient disk access.

Examples

Following are the examples, which demonstrate the use of dynamic memory allocation.

Example 1

In the following simple example we allocate a memory which is pointing by a character pointer. We copy an array of characters to that location and display it. After that we free that memory before exiting the program.

```
//This program allocates memory dynamically and then frees it after use.

#include                                    <iostream.h>
#include                                    <stdlib.h>
```

```
#include <string.h>

int main()
{
        char s1[] = "This is a sentence";
        char *s2;
        s2 = (char *) malloc(strlen(s1) + 1);
      /* Remember that stings are terminated by the null terminator, "\0',
           and the strlen returns the length of a string not including the terminator */
        if (s2 == NULL)
        {
                cout << "Error on malloc";
        return 1;
        /* Use a nonzero return to indicate an error has occurred */
        }

        strcpy(s2,s1);

        cout << "s1: " << s1 << endl;
        cout << "s2: " <<  s2 << endl;
        free(s2);
        return 0;
}
```

The output of the program is given below.

```
S1: This is a sentence
S2: This is a sentence
```

Example 2

Following is another example that allocates a memory dynamically according to the requirement and displays a message for the failure or success of the memory allocation.

```
// This program shows the dynamic allocation of memory according to the
requirement to //store a certain number of a structure.

#include <iostream.h>
#include <stdlib.h>
#include <string.h>

struct Employee
{
        char name[40];
        int id;
};

int main()
{
```

vubookshoppk.com

```
        Employee *workers, *wpt;
        int num;
        cout <<"How many employees do you want\n“ ;
        cin >> num;
        // the pointer workers gets the starting address of the memory if allocated
successfully
        workers = (Employee *) malloc(num * sizeof(Employee));
        if (workers == NULL)
        {
                cout << "Unable to allocate space for employees\n";
        return 1;
        // A nonzero return is usually used to indicate an error
        }
        cout << “Memory for “ << num << “ employees has allocated successfully” ;
        //now free the allocated memory
        free(workers) ;
}
```

A sample output of the program is as below.

```
How many employees do you want
235
Memory for 235 employees has allocated successfully
```

vubookshop.com

Exercise

As an exercise, you can find the maximum available memory from the heap on your computer. You can do this by using a loop in which first time you allocate a certain number of bytes(say 10000). If it is successfully allocated then free it and in the next iteration allocate twice of the previous size of memory. Thus we can find the maximum amount of memory available. Suppose you find that 2MB memory is available. Then run some other applications like MS WORD, MS EXCEL etc. Now again run your program and find out the size of the memory available now. Is there any difference in the size of the memory allocated? Yes, you will see that the size has decreased. It proves that the heap is being shared between all of the programs running on that machine at that time.

Dynamic memory allocation is a very efficient usage of computer resources as oppose to static memory allocation. The benefit of static memory is that its usage is very neat and clean, there are no errors. But disadvantage is that there are chances of wastage of resources.

The dynamic memory allocation is very efficient in terms of resources but added baggage is that freeing the memory is necessary, pointers management is necessary. You should avoid the situations that create memory leakage and dangling pointers.

Tips

- Using dynamic memory is more efficient then the static memory.
- Immediately after a memory allocation call, check whether the memory has allocated successfully.
- Whenever possible free the allocated memory in the same function.
- Be careful about memory management to prevent memory leakage and

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

dangling pointers.

- Before exiting the program, make sure that the allocated memory has freed.

vubookshoppk.com

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Lecture No. 25

Reading Material

Deitel & Deitel - C++ How to Program ***Chapter 3***
3.16, 3.18, 3.20

Summary

- Lecture Overview
- History of C/C++
- Structured Programming
- Limitations of Structured Programming
- Default Function Arguments
- Example of Default Function Arguments
- Placement of Variable Declarations
- Example of Placement of Variable Declarations
- Inline Functions
- Example of Inline Functions versus Macros
- Function Overloading
- Example of Function Overloading

vubookshoppk.com

Lecture Overview

From this lecture we are starting exciting topics, which we have been talking about many times in previous lectures. Until now, we have been discussing about the traditional programming following top down approach using C/C++. By and large we have been using C language, although, we also used few C++ functions like C++ I/O using **cin** and **cout** instead of standard functions of C i.e., **printf()** and **scanf**(). Today and in subsequent lectures, we will talk about C++ and its features. Note that we are not covering Object Oriented Programming here as it is a separate subject.

History of C/C++

C language was developed by scientists of Bell Labs in 1970s. It is very lean and mean language, very concise but with lot of power. C conquered the programming world and took it by storm. Major operating systems e.g., Unix was written in C language.

Going briefly into the history of languages, after the Machine Language (language of 0s and 1s), the Assembly Language was developed. Using Assembly language, programmers could use some symbolic codes, which were easier to understand by novice people. After that high-level languages like COBOL, FORTRAN were developed. These languages were more English like and as a result easier to understand for us as human beings. This was the age of spaghetti code where programs were not properly structured and their branches were growing in every direction. As a result, it is difficult o read, understand and manage. These problems lead to the innovation of structured programming where a problem was broken into smaller parts. But this approach also had limits. In order to understand those limits, we will see what is structured programming first before going into its limitations detail.

Structured Programming

We have learned so far, C is a language where programs are composed of functions. Basically, a problem is broken into small pieces or modules and each small piece corresponds to a function. This was the **top-down structured programming** approach. We have already discussed few rules of structured programming, which are still valid and will remain valid in the future. Let's reiterate those:

- **Divide and Conquer**; one should not write very long functions. If a function is getting longer than two or three pages or screens then it is divided into smaller, concise and well-defined tasks. Later each task becomes a function.
- Inside the functions, **Single Entry Single Exit** rule should be tried to obey as much as possible. This rule is very important for readability and useful in managing programs. Even if the developer itself tries to use the same function after sometime, it would be easier for him to read his own code if he has followed the rules properly. We try to reuse our code as much as possible. It is likely that we may reuse our code or functions. That reuse might happen quite after sometime. Never think that your written code will not change or will not be used again.
- You should **comment your programs** well. Your comments are only not used by other people but by yourself also, therefore, you should write useful and lots of comments. At least comment, what the function does, what are its parameters and what does it return back. The comments should be meaningful and useful about the processing of the function.

You should use the principles of structured programming as the basis of your programs.

Limitations of Structured Programming

When we design a functional program, the data it requires to process, is an entity that lies outside of the program. We take care of the function rather than the data it is

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

going to process. When the problems became complex, we came to know that we can't leave the data outside. Somehow the data processed by the program should be present inside it as a part of it. As a result, a new thought process became prevalent that instead of the program driven by functions, a program should be driven by data. As an example, while working with our Word processors when we want a text to be bold, firstly that text is selected and then we ask Word to make it bold. Notice in this example the data became first and then the function to make it bold. This is programming driven by data. This approach originated the Object Oriented Programming.
In the early 1980s a scientist in Bell Labs Bejarne Stroustrup started working in enhancing C language to overcome the shortcomings of structured approach. This evolution of C language firstly known to be **C with Classes,** eventually called **C++**. Then the follow-up version of C++ is the Java language. Some people call Java as C plus plus minus. This is not exactly true but the evolution has been the same way.
C++ does not contain the concept of Classes only but some other features were also introduced. We will talk about those features before we talk about the classes.

Default Function Arguments

While writing and calling functions, you might have noticed that sometimes the parameter values remain the same for most of the calls and others keep on changing. For example, we have a function:

```
power( long x, int n )
```

Where x is the number to take power of and n is the power to which x is required to be raised.

Suppose while using this function you came to know that 90% of the calls are for squaring the number x in your problem domain. Then this is the case where default function arguments can play their role. When we find that there are some parameters of a function that by and large are passed the same value. Then we start using default function arguments for those parameters.
The default value of a parameter is provided inside the function prototype or function definition. For example, we could declare the default function arguments for a function while declaring or defining it. Below is the definition of a very simple function **f()** that is called most of the times with parameters values of i as 1 and x as 10.5 most of the times then by we can give default values to the parameters as:

```
void f ( int  i = 1, double  x = 10.5 )
{
        cout  <<  "The value of i is: "  <<  i;
        cout  <<  "The value of x is: "  <<  x;
}
```

Now this function can be called 0, 1 or 2 arguments.
Suppose we call this function as:

```
f();
```

See we have called the function **f()** without any parameters, although, it has two parameters. It is perfectly all right and this is the utility of default function arguments. What do you think about the output. Think about it and then see the output below:

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
The value of i is: 1
The value of x is: 10.5
```

In the above call, no argument is passed, therefore, both the parameters will use their default values.
Now if we call this function as:

```
f(2);
```

In this case, the first passed in argument is assigned to the first variable (left most variable) **i** and the variable **x** takes its default value. In this case the output of the function will be as under:

```
The value of i is: 2
The value of x is: 10.5
```

The important point here is that your passed in argument is passed to the first parameter (the left most parameter). The first passed in value is assigned to the first parameter, second passed in value is assigned to the second parameter and so on. The value **2** cannot be assigned to the variable x unless a value is explicitly passed to the variable **i**. See the call below:

```
f(1, 2);
```

The output of the function will be as under:

```
The value of i is: 1
The value of x is: 2
```

Note that even the passed in value to the variable **i** is the same as its default value, still to pass some value to the variable **x**, variable **i** is explicitly assigned a value.

While calling function, the arguments are assigned to the parameters from left to right. There is no luxury or feature to use the default value for the first parameter and passed in value for the second parameter. Therefore, it is important to keep in mind that the parameters with default values on left cannot be left out but it is possible for the parameter with default values on right side.
Because of this rule of assignment of values to the parameters, while writing functions, the default values are written from right to left. For example, in the above example of function **f()**, if the default value is to be provided to the variable **x** only then it should be on the left side as under:

```
void f( int  i, double  x = 10.5 )
{
        // Display statements
}
```

If we switch the parameters that the variable **x** with default value becomes the first parameter as under:

```
void f( double  x = 10.5, int  i )
{
        // Display statements
}
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Now we cannot use the default value of the variable **x**, instead we will have to supply both of the arguments. Remember, whenever you want to use default values inside a function, the parameters with default values should be on the extreme right of the parameter list.

Example of Default Function Arguments

```
// A program with default arguments in a function prototype

#include <iostream.h>
void show( int = 1, float = 2.3, long = 4 );
void main()
{
  show();              // All three arguments default
  show( 5 );           // Provide 1st argument
  show( 6, 7.8 );      // Provide 1st and 2nd
  show( 9, 10.11, 12L ); // Provide all three argument
}
void show( int first, float second, long third )
{
  cout << "\nfirst = "  << first;
  cout << ", second = " << second;
  cout << ", third = "  << third;
}
```

The output of the program is:

```
first = 1, second = 2.3, third = 4
first = 5, second = 2.3, third = 4
first = 6, second = 7.8, third = 4
first = 9, second = 10.11, third = 12
```

www.vubookshoppk.com

Placement of Variable Declarations

This has to do with the declaration of the variables inside the code. In C language, all the variables are declared at the top of the function or code block and then we can use them later on in the code. We have already relaxed this rule, now, we will discuss it explicitly.

One of the enhancements in C++ over C is that a variable can be declared anywhere in the function. The philosophy of this enhancement is that a variables is declared just before it is actually used in the code. That will increase readability of the code.

It is not hard and fast direction but it is a tip of good programming practice. One can still declare variables at the start of the program, function or code block. It is a matter of style and convenience. One should be consistent in his/her style.

We should be clear about implications of declaring variables at different locations. For example, we declare a variable **i** as under:

```
{       // code block
        int  i;
```

```
        . . .
        . . .
    }
```

The variable **i** is declared inside the code block in the beginning of it. **i** is visible inside the code block but after the closing brace of this code block, **i** cannot be used. Be aware of this, whenever you declare a variable inside a block, the variable **i** is alive inside that code block. Outside of that code block, it is no more there and it can not referenced any further. Compiler will report an error if it is tried to access outside that code block.

You must have seen in your books many times, a for loop is written in the following manner:

```
for (int  i = 0; condition; increment/decrement statements )
{
        . . .
}
i = 500;          // Valid statement and there is no error
```

The variable **i** is declared with the for loop statement and it is used immediately. We should be clear about two points here. Firstly, the variable **i** is declared outside of the for loop opening brace, therefore, it is also visible after the closing brace of the for loop.
So the above declaration of i can also be made as under:

```
int  i;
for ( i = 0; condition; increment/decrement statements)
{
        . . .
}
```

This approach is bit more clear and readable as it clearly declares the variable **i** outside the for statement. But again, it is a matter of style and personal preference, both approaches are correct.

Example of Placement of Variables Declarations

```
// Variable declaration placement
#include  <iostream.h>

void  main()
{
//   int  lineno;
   for( int lineno = 0; lineno < 3; lineno++ )
   {
      int  temp = 22;
      cout << "\nThis is line number " << lineno
           << " and temp is " << temp;
   }
```

vubookshoppk.com

```
  if( lineno == 4 )   // lineno still accessible
     cout << "\nOops";
  // Cannot access temp
}
```

The output of the program is:

```
This is line number 0 and temp is 22
This is line number 1 and temp is 22
This is line number 2 and temp is 22
```

Inline Functions

This is also one of the facilities provided by C++ over C. In our previous lectures, we discussed and wrote macros few macros like **max** and **circlearea**.
While using macros, we use the name of the macro in our program. Before the compilation process starts the macro names are replaced by the preprocessor with their definitions (defined with #define).
Inline functions also work more or less in the same manner as macros. The functions are declared inline by writing **inline** keyword before the name of the function. This is a directive to the compiler and it causes the full definition of the function to be inserted in each place the function is called. Inserting individual copies of functions eliminates the overhead of calling a function (such as loading parameters onto the stack).

We see what are the advantages and disadvantages of it:

We'll discuss the disadvantages first. Let's suppose the inline function is called 100 times inside your program and that function itself is of 10 lines in length. Then at 100 places inside your program this 10 lines function definition is written, causes the program size to increase by 1000 lines. Therefore, the size of the program increases significantly. The increase in size of program may not be an issue if you have lots of resources of memory and disk space available but preferably, we try not to increase the size of the program without any benefit.

Also the inline directive is a request to the compiler to treat the function as inline. The compiler is on its own to accept or reject the request of inlining. To get to know whether the compiler has accepted the request to make it inline or not, is possible through the program's debugging. But this is bit tedious at this level of our programming expertise.

Now we'll see what are the advantages of this feature of C++. While writing macros, we knew that it is important to enclose the arguments of macros within parenthesis. For example, we wrote square macro as:

```
#define  square(x)      (x)  *  (x)
```

when this macro is called by the following statement in our code:

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
square( i + j );
```

then it is replaced with the definition of the square macro as:

```
( i + j ) * ( i + j );
```

Just consider, we have not used parenthesis and written our macro as under:

```
#define  square(x)      x  *  x
```

then the substitution of the macro definition will be as:

```
i + j  *  i + j;
```

But the above definition has incorrect result. Because the precedence of the multiplication operator (*) is higher than the addition operator (+), therefore, the above statement is executed semantically as:

```
i + (j * i) + j;
```

Hence, the usage of brackets is necessary to make sure that the macros work as expected.

Secondly, because the macros are replaced with preprocessors and not by compiler, therefore, they are not aware of the data types. They just replace the macro definition and there is no type checking on the parameters of the macro. Same macro can be used for multiple data types. For instance, the above square macro can be used for **long**, **float**, **double** and **char** data types.

Inline functions behave as expected like a function and they don't have any side effects. Secondly, the automatic type checking for parameters is also done for inline functions. If there is a difference between data types provided and expected, the compiler will report an error unlike a macro.

Now, we see a program code to differentiate between macros and inline functions:

Example of Inline Functions versus Macros

```
// A macro vs. an inline function

#include  <iostream.h>

#define  MAX( A, B )  ((A) > (B) ? (A) : (B))
inline  int  max( int a, int b )
{
   if ( a > b )
         return a;
   return b;
}

void main()
{
  int  i, x, y;
  x = 23; y = 45;
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
  i = MAX( x++, y++ );  // Side-effect:
                        // larger value incremented twice
  cout << "x = " << x << "    y = " << y << '\n';

  x = 23; y = 45;
  i = max( x++, y++ );  // Works as expected
  cout << "x = " << x << "    y = " << y << '\n';
}
```

The output of this program is:

```
x = 24   y = 47
x = 24   y = 46
```

You can see that the output from the inline function is correct while the macro has produced incorrect result by incrementing variable y two times. Why is this so?
The definition of the macro contains the parameters A and B two times in its body and keeping in mind that macros just replace the argument values inside the definition, it looks like the following after replacement.

```
( (x++) > (y++)  ?  (x++)  :  (y++) );
```

Clearly, the resultant variable either **x** or **y**, whichever is greater (**y** in this case) will be incremented twice instead of once.

Now, the interesting point is why this problem is not there in inline functions. Inside the code, the call to the inline function **max** is made by writing the following statement:

```
i  =  max( x++, y++ );
```

While calling the inline function, compiler does the type checking and passes the parameters in the same way as in normal function calls. The arguments are incremented once after their values are replaced inside the body of the function **max** and this is our required behavior.

Hence, by and large it is better to use inline functions rather than macros. Still macros can be utilized for small definitions.

The inline keyword is only a suggestion to the compiler. Functions larger than a few lines are not expanded inline even if they are declared with the inline keyword.

If the inline function is called many times inside the program and from multiple source files (until now, usually we have been using only one source file) then the inline function is put in a header file. That header file can be used (by using #include) by multiple source files later.

Also keep in mind that after multiple files include the header file that contains the inline function, all of those files must be recompiled after the inline function in the header file is changed.

Now, we are going to cover exciting part of this lecture i.e., Function Overloading.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Function Overloading

You have already seen overloading many times. For example, when we used **cout** to print our string and then used it for **int**, **long**, **double** and **float** etc.

```
cout << "This is my string";
cout << myInt ;
```

This magic of **cout** that it can print variables of different data types is possible because of overloading. The operator of cout (<<) that is stream insertion operator is overloaded for many data types. Header file **iostream.h** contains prototypes for all those functions. So what actually is overloading?
"Using the same name to perform multiple tasks or different tasks depending on the situation."
cout is doing exactly same thing that depending on the variable type passed to it, it prints an **int** or a **double** or a **float** or a **string**. That means the behavior is changing but the function **cout <<** looks identical.
As we all know that computers are dumb machines and they cannot decide anything on their own. Therefore, if it is printing variables of different types, we have to tell it clearly and separately for each type like **int** or **double** etc. In this separately telling process, the operator used is the same **<<**. So in a way that operator of **<<** is being overloaded. For this lecture, we will not go into the detail of operator overloading but we will limit our discussion to function overloading.

Function overloading has the same concept that the name of the function will remain same but its behavior may change. For example, if we want to take square root of a number. That number can be an integer, float or a double and depending on the type of the argument, we may need to do different calculation. If we want to cater to the two data types **int** and **double**, we will write separate functions for **int** and **double**.

```
double intsqrt ( int i );
double doublesqrt ( double d );
```

We can use the function **intsqrt()** where integer square root is required and **doublesqrt()** where square root of double variable is required. But this is an overhead in the sense that we have to remember multiple function names, even if the behavior of the functions is of similar type as in this case of square root. We should also be careful about auto-widening that if we pass an **int** to **doublesqrt()** function, compiler will automatically convert it to **double** and then call the funtion **doublesqrt()**. That may not be what we wanted to achieve and there is no way of checking that we have used the correct function. The solution to this problem is function overloading.

While overloading functions, we will write separate functions for separate data types but the function name will remain same. Return type can be different if we want to change, for example in the above case we might want to return an **int** for square root function for **int**s and **double** for a square root of a **double** typed variable. Now, we will declare them as under:

```
int     sqrt ( int i );
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
double  sqrt ( double d );
```

Now, we have two functions with the same name. How will they be differentiated inside the program?

The differentiation comes from the parameters, which are passed to these functions. If somewhere in your program you wrote: **sqrt(10.5)**, the compiler will automatically determine that **10.5** is not an integer, it is either **float** or a **double**. The compiler will look for the **sqrt()** with parameter of type **float** or a parameter with type as **double**. It will find the function **sqrt()** with double parameter and call it. Suppose in the subsequent code, there is a call to **sqrt()** function as under:

```
int  i;
sqrt ( i );
```

Now, the compiler will automatically match the prototype and will call the **sqrt()** with **int** as parameter type.

What is the advantage of this function overloading?
Our program is more readable after using function overloading. Instead of having lot of functions doing the same kind of work but with different names. How does the compiler differentiate, we have already discussed that compiler looks at the type and number of arguments. Suppose there are two overloaded functions as given below:

```
int  f( int x, int y );
int  f( int x, int y, int z );
```

One function **f()** takes two **int** parameter and other one takes three **int** type parameters. Now if there is call as the following:

```
int  x = 10;
int  y = 20;
f( x, y );
```

The function **f()** with two int parameters is called.
In case the function call is made in the following way:

```
int  x = 10;
int  y = 20;
int  z = 30;
f( x, y, z );
```

The function **f()** with three **int** parameters is called.

We have not talked about the return type because it is not a distinguishing feature while overloading functions. Be careful about it, you cannot write:

```
int       f ( int  );
double  f ( int );
```

The compiler will produce error of ambiguous declarations.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

So the overloaded functions are differentiated using type and number of arguments passed to the function and not by the return type. Let's take a loop of some useful example. We want to write functions to print values of different data types and we will use function overloading for that.

```
/* Overload functions to print variables of different types */

#include <iostream.h>

void print (int i)
{
        cout << "\nThe value of the integer is: " << i;
}
void print (double d)
{
        cout << "\nThe value of the double is: " << d;
}

void print (char* s)
{
        cout << "\nThe value of the string is: " << s;
}


main (void)
{
        int i = 100;
        double d = 123.12;
        char *s = "This is a test string";

        print ( i );
        print ( d );
        print ( s );
}
```

vubookshoppk.com

The output of the program is:

```
The value of the integer is: 100
The value of the double is: 123.12
The value of the string is: This is a test string
```

You must have noticed that automtically the **int** version of **print()** function is called for **i**, **double** version is called for **d** and **string** version is called for **s**.
Internally, the compiler uses the **name mangling** technique to generate a unique token that is assigned to each function. It processes the function name and its parameters within a logical machine to generate this unique number for each function.

Example of Function Overloading

```
/* The following example replaces strcpy and strncpy with the single function name
       stringCopy. */

// An overloaded function
#include <iostream.h>
#include <string.h>

inline void stringCopy( char *dest, const char *src )
{
   strcpy( dest, src );          // Calls the standard C library function
}
inline void stringCopy( char *dest, const char *src, int len )
{
   strncpy( dest, src, len );  // // Calls another standard C library function
}

static char stringa[20], stringb[20];   // Declared two arrays of characters of size 20

void main()
{
   stringCopy( stringa, "That" );       // Copy the string 'That' into the array stringa
   stringCopy( stringb, "This is a string", 4 );  // Copy first 4 characters to stringb array
   cout << stringb << " and " << stringa;  // Display the contents on the screen
}
```

vubookshoppk.com

Lecture No. 26

Reading Material

Deitel & Deitel - C++ How to Program Chapter. 6

6.5, 6.7, 6.8, 6.10, 6.11, 6.14

Summary

- Classes and Objects
- Definition of a class
- Separation of Interface from the Implementation
- Structure of a class
- Sample program
- Constructor
- Default arguments with constructors
- Tips

Classes and Objects

In today's lecture, we will try to learn about the concepts of 'classes' and 'objects'. However, we are not going to formally cover the object-oriented programming but only the ways to manipulate the classes and objects.

We had talked about structures in our previous lectures. In structures, some data variables are gathered, grouped and named as a single entity. Class and structure are very closely related. In classes, we group some data variables and functions. These functions normally manipulate these variables.
Before going ahead, it is better to understand what a class is:

"A class includes both data members as well as functions to manipulate that data"

These functions are called 'member functions'. We also call them methods. So a class has data (the variables) and functions to manipulate that data. A class is a 'user defined' data type. This way, we expand the language by creating a new data type. When we create variables of a class, a special name is used for them i.e. Objects.

"Instances of a class are called objects"

With the definition of class, we have a new data type like int, char etc. Here *int i;* means 'i' is an instance of data type *int*. When we take a variable of a class, it becomes the instance of that class, called object.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Definition of a class

Let's have a look on the structure of a class. It is very similar to the struct keyword. Keyword class is used and the braces enclose the definition of the class i.e.

```
class name_of_class{

        // definition of class
}
```

The new data type i.e. classes helps us to have grouped data members and member functions to manipulate the data. Consider a structure of Date having data members i.e. year, month and day. Now we can declare a variable of structure Date and use dot operator to access its members i.e.

```
Date myDate;
myDate.month=3;
```

We have to use the name of the object, a dot operator and the data member of structure to be accessed. The data members are of normal data types like int, float, char etc. Other data types can also be used.

Let's consider an example of Date Class shown in the following statement.

```
class Date{

        int Day;
        int month;
        int year;
};
```

Now we will take its object in the fashion given below:

```
Date myDate;
```

Separation of Interface from the Implementation

To access the data members of the class, we will again use dot operator. Before going ahead, we will see what is the difference between struct and class. It is the visibility of the data members that differentiates between struct and class. What does the word 'visibility' mean? Consider an example of payroll system. We have stored the tax rate i.e. 5% in a variable *i* of type int. Later, we used the same *i* in a loop and changed the value of tax rate unintentionally. Now the calculation of the pay in the end will not provide the correct results. To avoid this problem, we can tag the tax rate variable as *int tax_rate*;. But this variable again is visible in the whole program and anyone can change its value. The data is open and visible to every part of the program, creating a big problem.

In normal programming, we will like to see the data encapsulated. It means that data is hidden somewhere. However, it can be used. Let's consider a real world problem to understand it. Most of us have wrist-watches. To have accuracy, it is necessary to adjust the time. How can we do that? We can change the time by using the button that is provided on one side of the watch. This is a kind of encapsulation. We can see the

vubookshoppk.com

hands of the watch but cannot touch them. To change their position we used the button. Whenever we talk about the class, we have to think of this concept that data is available somewhere. We don't need to know about the exact structure i.e. what is inside the watch. All we know is that its internal structure is defined somewhere that cannot be seen or touched. We can only see its interface. If we need to adjust the time, a button may be used. It is a nice separation of implementation and interface. Classes allow us to do that.

Structure of a class

Let's have a look inside a class. Consider the example of class Date. Can we set the values of the data members of the object 'myDate' i.e. day, month or year. We cannot say like *myDate.month = 11*;. Try to do this. The compiler will give error and stop compiling the program. It will not recognize the variable 'month'. In other words, it cannot see 'month'. The default visibility for the data members of the class is called 'private'. These can only be used within the class and are not visible outside.

"The default visibility of all the data members and member function of a class is
hidden and private"

'private' is also a keyword. What will be the opposite of the private? What we will have to do to use the data members and manipulate them. The keyword for this purpose is public. In the class definition, if you do not mention the visibility and start defining the data and functions, these will be by default private. As a good programmer, we should always write the keyword private with a colon as:

```
private:
```

Now all the data and functions following this statement will have the private visibility. To define the public data, we need to write the keyword public with a colon as:

```
public:
```

Now all the data and functions following the public keyword will have the public visibility. These will be visible from outside the class. We can have multiple public and private parts in the class definition but it becomes confusing. So normally we have only one public and one private part. Again consider the Date example. By making the data members as private, we will write functions to set and get the date. As this is needed to be visible from outside the class, these functions will be defined as public.

```
class Date
{
   private:
                  // private data and functions
   public:
                  // public data and functions
};
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Normally, the data in a class is kept private. If we make the data public, it is same as structure and anyone can access this data. On the other hand, the functions which we have written to manipulate this data, are kept as public. These methods can be called from outside the class i.e. from the main program. These are the member functions of the class. The difference between these and the ordinary functions is that they are part of class. Moreover, they can see the private data members of the class and also manipulate them.

We have made the data members private in the Date class. In the program, we take an object of Date class as *Data myDate*;. *myDate* is a variable of type Date. Now if we say *myDate.month = 3;* this statement will be illegal as the *month* is a private data member of the Date class. Now try to understand this concept. You can think class as a box having different things in it. How can we touch inside the box? We have a window and can see only those things that are visible through this window. Those things which we cannot see from the window, can not be accessed from outside. Day, month and year are somewhere inside the box and are not visible through the window. Now we want to assign some values to these data members. For this purpose, we will define a member function in the class in the public section. Being present in public section, it will be visible through the window. As this is the member function, it can see and manipulate the private data of the class. Now it's a two-step process. We can see the public functions and public functions can view the private data members of the class. We will write a function to set the value to the month. We cannot write it as *myDate.month = 10*;. So our function prototype will be as:

```
void setMonth(int month)
```

and we may call this function as:

```
myDate.setMonth(10);
```

Now the function *setMonth* will assign the value 10 to *month* data member of the object *myDate*. The same thing will be applicable if we want to print the date. We can write a public function print and can access it as:

```
myDate.print();
```

The function print can see the private data members. So it will format the date and print it. In structures, the data members are public by default. It means that these are visible to all and anyone can change them. Is there any disadvantage of this? Think about the date. What may be the valid values of the day? Can we have a day less than zero or greater than 32. So the minimum and maximum values of the day are 1 and 31 respectively. Similarly, in case of month, the minimum and maximum values may be 1 and 12. We can assign different values to year like 1900, 2002 etc. If we are using Date structure instead of a class, we can write in the program as *myDate.month=13;* and the month will be set to 13. So the date will become invalid. We may want that other programmers also use this structure. But other programmers may put invalid values to the data-member as these are publicly accessible. Similarly in structures, everything is visible i.e. what are the names of the data members. How are these manipulated?

vubookshoppk.com

Now we want that only those things should be visible which we want to show and those things which we want to hide should not be visible We can get this by using the private and public in the classes. Public becomes the interface of the class, what we want to show to others. With the use of public interface, the objects can be manipulated. Private becomes the inside of the class i.e. the data members, the implementation. We don't want to show the implementation of our classes to others. This is the concept of separation of interface from implementation. It is a crucially important concept in modern programming. We have separated the interface from the implementation. As long as the interface remains the same, the implementation can be changed. Let's think about it in real world. The example from the automobiles sector can help us understand further. The production of cars in the world started in the late 18th century and early 19th century. Let's compare these early or prototype cars with today's modern ones. There is a big difference between the old and new cars. Technology has changed. Now what is still common in both the types. Steering, clutch, brakes and accelerator pads are still the basic components of a car. So the interface is same. The internal functionality can be changed. To turn the car, old cars used rod mechanisms and modern cars have the microprocessor to do this job. Our physical action is same in both the cases. The interface i.e. steering is same and also the effect that wheels have turned to right is the same too. The internal implementation has completely changed. The old combustion engine cannot be compared with the state-of-the technology based modern engines. But the interface is the same i.e. we turn the key to start an engine. This concept of separation of implementation from interface comes into our programming. We have written a program today to calculate the orbital time of moon around the earth. In today's physics, we have formula to calculate this. We have defined the interface *calculateOrbitalTime()*. This is a function that will calculate the orbital time of moon around earth. This formula may prove wrong after some time. Now what can we do? Despite the change in the implementation, interface remains the same i.e. the name of the function is same. Now when the program will use this function, it gets the correct result as we have implemented the new formula inside the function. Moreover, the main program does not need to be changed at all. Being a very neat concept, it can be used while dealing with objects and classes.

Sample program

Let's see the example of Date class in detail.

```
class Date
{
   public:
                  void display();
                  Date(int, int, int);
   private:
                  int day, month, year;
};
```

Date is the name of new user defined data type. After the braces, we have written the keyword public. In this section, we will define the interface of the class. We have declared a function *display()* which will print the date on the screen. Another function *Date(int day, int month, int year)* is declared. The name of this function is same as the name of the class, having no return type. This function is called constructor. Then we

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

write the keyword private and define the implementation of the class. Here we have three variables i.e. day, month and year of type int. In the end closing braces and the semi-colon. This is the definition of user defined data type i.e. class. It will not occupy any memory space as it has no data currently. It is the same as we write in case of 'int'. It does not occupy any memory but when we say *int I*, the memory is reserved for *i*. The class is a 'user defined data' type. Now in our program, when we write *Date myDate*; an instance of the class is created i.e. an object. Object reserves space in the memory. Object will have these data members. What about the 'functions'? For a moment, we can say that functions are also in the memory.

We want to use this class in our program and display the date using the *display()* function. We have written the prototype of the *display()* function in the class without defining the *display()* function yet. A special way is used to define these functions. We will write the name of the class, followed by two colons and the name of the function. The rest is same as we used to do with ordinary functions.

```
Date::display()
{
        // the definition of the function
        cout << "The date is " << day << "-" << month << "-" << year <<
endl;
}
```

You might have noted the difference in the first line. The double colon is called scope resolution operator. It resolves the scope and tells that this function belongs to whom. In this case, the (Date::display())) tells that the *display()* function belongs to the Date class. So the scope resolution is required. In a way, consider it as function is defined inside the class. If you have private function, even then the definition mechanism is same. We will define the function outside of the class. Even then it will not be visible as its visibility is private. The way to define the member functions is, class name, double colon, name of the function including arguments and then the body of the function. Can we define the function inside the class? Yes we can. When we write the function inside the class, the compiler tries to treat that function as inline function. As a good programming practice, we define the functions outside of the class. So to make sure that the function belongs to the class, the scope resolution operator is used.

We have so far tried to discuss Date class at a rudimentary level. That is we can create objects of Date class and display the date using its functions. We can do a lot of other things with this class. When we say *int i;* and ask to print its value. The answer is that we have not assigned any value to it yet and don't know what will be there at that memory location. Similarly, when we declare an object of the Date class as *Date myDate*; an object is created. But we don't know about the values of day, month and year. Now if we call its public function *display()* using the dot operator as *myDate.display()*. It will print whatever the value is in the data members. We need functions to set/change the date. Suppose we want to set the day, the month and the year separately. For this purpose, we need three more public functions. We can name these functions as setDay(int), setMonth(int) and setYear(int). These functions may be called inside the program as *myDate.setDay(15)*, *myDate.setMonth(12)* and *setYear(2002)*. These functions will change the value of day, month and year. As these are member functions, so scope resolution operator is being used.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
void Date::setDay(int i)
{
        day = i;
}

void Date::setMonth(int i)
{
        month = i;
}

void Date::setYear(int i)
{
        year = i;
}
```

The question arises, which objects data members are being set. In the *setDay* function, we are assigning value to the *day*. But this *day* belongs to which object. The answer is, we have just defined the function, it is not called yet. The functions are called by the objects, not by the class. When we say *Date myDate*; it means that we have an object of type Date. Now we can say *myDate.setDay(10)*. The value of day of *myDate* object will be set to 10. When we create objects, these will reserve space in memory. Suppose, the objects are date1, date2, date3. These will be created at different memory locations having there own data members. When we call a member function with the object name, this function will manipulate the data of this object. Let's consider the following code snippet to understand it.

```
Date date1, date2, date3;
// Manipulating date1 object
date1.setDay(10);
date1.setMonth(12);
date1.setYear(2002);
date1.display();

// Manipulating date2 object
date2.setDay(15);
date2.setMonth(1);
date2.setYear(2003);
date2.display();
```

We have declared three objects of type Date. All these objects have data members day, month and year. When we call a function, that is defined in class, with some object name, it uses the data of that object which is calling the function. Suppose, when we write *date1.setMonth(12);* it will manipulate the data of object *date1*. Similarly when we say *date2.display()*, the function is defined inside the class. However, it will use the data of *date2* object. Remember that we will always call these member functions by referring to some specific object. We can call these functions with date1, date2 or date3 respectively. We will never call these functions referring to a class that is we cannot say *Date.display();* It is illegal. The functions of getting data from objects and setting data of objects are standard. So we normally use

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

the word ‘set’ for setting the data and ‘get’ for getting the data. It is a matter of style. You can call it whatever you want. But it will be a bad idea to name a function print() and it is setting the value of month. It will work but in a very confused manner. If we want to write a function to set month, the logical choice of the function name is *setMonth(int)*. Similarly, *setDay(int)* and *setYear(int)* will be used to set the day and year respectively. If we want to get the values of these data members, the logical choice will be *getDay(), getMonth()* and *getYear()*. The names are self-explanatory. These functions are defined as member functions of the class. They are put in the public section of the class and constitute the public interface of the class. These will be visible from outside the class. Normally they manipulate the data that is hidden inside the class i.e. in the private section of the class. No need to show the working of the functions only its name, argument and the return type is told to the user. User of the class is our program.

Here is the complete code of the Date class.

```
/* A sample program with the Date class. Set methods are given to set the day, month
and year.The date is also diplayed on the screen using member function. */

#include <iostream.h>

// defining the Date class
class Date{
   // interface of the class
   public:
      void display();       // to display the date on the screen
      void setDay(int i);   // setting the day
      void setMonth(int i); // setting the month
      void setYear(int i);  // setting the year

   // hidden part of the class
   private:
      int day, month, year;
};

// The display function of the class date
void Date::display()
{
        cout << "The date is " << day << "-" << month << "-" << year << endl;
}
// setting the value of the day
void Date::setDay(int i)
{
        day = i;
}

// setting the value of the month
void Date::setMonth(int i)
{
        month = i;
```

vubookshoppk.com

```
}

// setting the value of the year
void Date::setYear(int i)
{
        year = i;
}
// Main program. We will take two date objects, set day, month, year and display the
date.
int main()
{
   Date date1,date2;   // taking objects of Date class

   // setting the values and displaying
   date1.setDay(1);
   date1.setMonth(1);
   date1.setYear(2000);
   date1.display();

   // setting the values and displaying
   date1.setDay(10);
   date1.setMonth(12);
   date1.setYear(2002);
   date1.display();
}
```

The output of the program is:

```
The date is 1-1-2000
The date is 10-12-2002
```

Constructors

We have written a function named *Date(int, int, int)* in our class. This is in the public section of our class. It has no return type, having the name as that of class. Such functions are called constructors. When we create an object by writing *Date myDate;* A function is invisibly called which does something with this object. This function is constructor. If we do not write a constructor, C++ writes a default constructor for us. By and large, we want that the object should be created in a certain state. When our object *myDate* is created its data members-day, month and year have some value. We can initialize these data members with zero or with some specific date. How can we do that? Native data types can be initialized as:

```
        int i;
        i = 10;
OR
        int i = 10;
```

Generally, a constructor initializes the object into a state that is recognizable and acceptable. The default constructor does not take any parameter. We can have many constructors of a class by overloading them. The constructor for Date class is:

vubookshoppk.com

```
Date(int, int, int);
```

This is the prototype of the constructor that is defined in the class. The definition of constructor is same as we used with the member functions.

```
Date::Date(int theDay, int theMonth, int theYear)
{
        day = theDay;
        month = theMonth;
        year = theYear;
}
```

How can we call this constructor? We know that constructor is automatically called when an object is created. To use this constructor, we will take an object as:

```
Date myDate(1, 1 , 2003);
```

Here two things have taken place. 1) An object is created 2) The data members are initialized. This is happening in the memory at run time. Nothing will happen at compile time. The constructor will be called after the object creation and before the control given back to the program. Here the value of day of the *myDate* object is 1, the value of month is 1 and the value of year is 2003. It has created and initialized an object. Now if we call the *display()* function. These values will be displayed. Constructor is used to initialized an object and put it into a consistent and valid state.

Default arguments with constructors

We can also use the default arguments with the constructors. In the case of Date, normally the days and months are changing and the year remains same for one year. So we can give the default value to year.

```
Date::Date(int theDay, int theMonth, int theYear = 2002)
{
        // The body of the constructor
}
```

Now we have different ways of creating objects of class Date.

```
Date myDate;
```

In this case, the default constructor will be called while the data members remain un-initialized.

```
Date myDate(1, 1, 2000);
```

The constructor will be called and initialized. The day will be 1, month will be 1 and the year will be 2000.

```
Date myDate(1, 1);
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

The constructor will be called and initialized. The day will be 1, month will be 1 and the year will be initialized to the default value i.e. 2002.

There are some complications. Constructor is itself a function of C++ and can be overloaded. We can have many constructors. Suppose, we are asked to write the date i.e. 1, 1, 2000. Some of us may write it as 1, 1, 2000. Some will write it as 1/1/2000. A considerable number may write as 1-1-2000. One can write date as 1 Jan. 2000. There may have many formats of dates. It will be nice if we can initialize the object using any of these formats. So we may have a constructor which takes a character string. The date format is '01-Jan-2003'. So the constructor should parse the string. The string before the hyphen is day (i.e. 01) convert it into an integer and assign it to day. Again get the strings before the 2nd hyphen (i.e. Jan), check which month is it (i.e. 1) and assign it to month. Rest of the string is year so convert it into integer and assign it to year. We are doing a lot of horizontal integration here. The good thing is that the rules of simple functions overloading applies to constructors also. The rules of default arguments also apply while we are using default arguments with constructors. The idea is to make the class as friendly as possible for the users. We have two constructors. Of these, one takes three ints and the other takes the date as a character string. We may want to add more constructors. But we don't want to add too many constructors in the class as there is a limit of everything. Within limits and the reasons, provision of two to three alternatives to the users of the class for object creation is nice. May be the program that is using our class, is applying months as character strings. We should provide a constructor that deals with this. We will further explain this subject in the coming lectures. A constructor is a special kind of function having same name as that of a class. It has no return type. Declare it without return type. Constructor can take arguments. The default constructor takes no argument.
Here is the code of the Date class using the different constructors.

```
/*
A sample program with the Date class. Use of constructors is shown here.
*/

#include <iostream.h>
//#include <stdlib.h>

// defining the Date class
class Date{
   // interface of the class
   public:
      void display();       // to display the date on the screen
      void setDay(int i);   // setting the day
      void setMonth(int i); // setting the month
      void setYear(int i);  // setting the year
      int getDay();         // getting the value of day
      int getMonth();       // getting the value of month
      int getYear();        // getting the value of year

      // Constructors of the class
      Date();
      Date(int, int, int);
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
    // hidden part of the class
    private:
        int day, month, year;
};

// defining the constructor
// default constructor. setting the date to a default date

Date::Date()
{
    day = 1;
    month = 1;
    year = 1900;
}

// Constructors with default arguments

Date::Date(int theDay, int theMonth, int theYear = 2002)
{
        day = theDay;
        month = theMonth;
        year = theYear;
}

// The display function of the class date
void Date::display()
{
        cout << "The date is " << getDay() << "-" << getMonth() << "-" << getYear()
<< endl;
}
// setting the value of the day
void Date::setDay(int i)
{
        day = i;
}

// setting the value of the month
void Date::setMonth(int i)
{
        month = i;
}

// setting the value of the year
void Date::setYear(int i)
{
        year = i;
}
// getting the value of the day
int Date::getDay()
```

vubookshoppk.com

```
{
        return day;
}

// getting the value of the month
int Date::getMonth()
{
        return month;
}

// getting the value of the year
int Date::getYear()
{
        return year;
}


// Main program. We will take three date objects using constructors, and display the
date.
int main()
{
   Date date1, date2(1, 1, 2000), date3(10,12);   // taking objects of Date class

   // displaying the dates on the screen
   date1.display();
   date2.display();
   date3.display();

}
```

vubookshoppk.com

The output of the program is:

```
The date is 1-1-1900
The date is 1-1-2000
The date is 10-12-2002
```

Summary

A class is a user defined data type. It has data members and member functions. Normally member functions are called methods. Data members are generally kept as private. The member functions, used to manipulate the data members, are kept public so that these are visible from outside the class. The public part of the class is known as the interface of the class. It may contain data members and functions but normally we put functions as public. The member functions can manipulate the data members (public and private) of the class. Non-member functions can not see or access the private part of the class. We try to separate the implementation of the class from its interface.

Tips

- Explicitly write keyword private in the class definition

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

- Separate the interface and implementation
- The default constructor has no arguments
- Constructor has the same name as of class
- The data members of the class are initialized at runtime
- Initializing the data members in the definition of the class is a syntax error

vubookshoppk.com

Lecture No. 27

Reading Material

Deitel & Deitel - C++ How to Program Chapter. 6

6.9, 6.10, 6.11, 6.12, 6.13, 6.14

Summary

1) Classes And Objects
2) Constructors
3) Types of Constructors
4) Utility Functions
5) Destructors

Classes and Objects

This lecture is a sequel of the previous discussion on 'Classes' and 'Objects'. The use of 'classes and objects' has changed our way of thinking. Instead of having function-oriented programs i.e. getting data and performing functions with it, we have now data that knows how to manipulate itself. This way, now the programming is object-oriented. It means that our programs revolve around data and the objects. Therefore, it would be nice to have some building blocks for programs so that these can be combined to write a program. We have so far talked about simple variables like integer, double and char, followed by strings of characters and arrays of different data types. But now, in the form of an object, we have a block which knows itself about contents and the behavior. The upcoming discussion will further explain it. We have used *cout* for displaying many things like integers, doubles and strings. Here integer did not know how it is going to display itself. However, *cout* knows how to display an integer on the screen. Now we want to see that an integer should know how to display itself. So it will be a different process. Now the question arises whether it will be good to see an integer knowing how to display itself? For this purpose, we will have to expand the scope of thinking.

While engaged in the process of programming, we try to solve a real-world problem. The real world is not only of integers, floats, doubles and chars, but there are other things like cycles, cars, buildings, schools and people. We perceive all these things as objects. Each object has a behavior associated with it. Consider the example of a man who can talk, walk, sit and stand etc. Similarly, we can think of a vehicle that has many functions. These objects also have attributes. For example, a man has height, weight, color of eyes and hair and so on. These all are his attributes. His actions will be referred as functions or methods. This principle may be applicable to vehicles, aeroplanes and all other real-world things. An aeroplane has attributes like its height, width, number of seats and number of engines etc. These are attributes called data members. Its actions include takeoff, flying and landing. These actions are its functions or methods.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

In terms of language, the attributes and actions may be equated with nouns and verbs. The verbs are actions which are also called methods in the programming terminology. These methods should be included in the object in such a way that the object itself knows how to achieve this function or method. Now consider this all in terms of data. Is it always in terms of salary, payroll, amount and numbers? Actually, data comes in different varieties. Now a day, a computer is a multimedia equipment. We see that there are not only alphabets and digits displayed on the screen, but also pictures, images, windows, dialogue boxes, color and so many other things. These are numbers, letters and graphics. Other than this, we can see videos on our computer. It is just another type of media. Similarly, we find audio material, which can be played on the computer. Thus in the expanded terms of data, we come across numbers, pictures, audio and video while dealing with multimedia.
Now we think about the concept that an integer should know how to display itself. With the enhanced scope of data, we can also have an audio, which knows how to play itself. The same applies to video.

Class

A class is a way of defining a user-defined data type. In a class, one may find data members and functions that can manipulate that data. In the previous lectures, we have talked about the concept of data hiding i.e. encapsulation that means that the data of a class cannot be accessed from outside. However, it can be done through some defined functions (methods). These are the member functions of the class. To hide the data, we declare it *private*. If a data is private, it will be available only to member functions of the class. No other function outside the class (except friend functions) can access the private data. Normally in a class we divide the private part which is normally what we called implementation of the class, from the functions that manipulate that private data which is called the interface (which is the front end).
The example of a room can help us understand private and public parts of a class. A class is a room having a curtain in its middle. The things behind the curtain (private) are visible to the residents (insiders) of the room. They know about every thing present in the room. When the door opens, the outsiders see only the things in front of the curtain. This is the public interface of the class while behind the curtain is the private interface. A function inside the class (i.e. a member function) can access and manipulate all things in the class. A function outside the class can only access and manipulate its public interface part. A constructor has to be in the public section of the class. There should also be a public interface so that it can be called from outside.

Constructors

Constructor is a special function, called whenever we instantiate an object of a class. If we do not define a constructor function in a class, the C++ provides a default constructor. It is executed at the time of instantiating an object.
To understand the basic function of constructor, we have to go back. While writing c++ Stroustrup noticed that the majority of programming problems, which we call bugs, occur due to the use of uninitialized data. That is, we declare variables and use them without providing them any value. For example, we declare an integer as *int i ;* And it is not initialized with a value like *i= 0; or i = 5;* And then somewhere in the

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

program, we write, say, *j = 2 * i ;*. This is the usage of an uninitialized data variable. This technique has a demerit that despite having no syntax error, it may cause a logical error, which is difficult to find. Thus, initialization of data is a very critical activity. The constructors give us an opportunity to initialize the data members of an object in such a way that when our program gets an object, the data part of the object is in a known state. Being in a valid state, it can be used. The constructors are used to initialize the data members of an object. A class is a user defined data type it does not take space in the memory unless we create an object from it. The constructors create space for data members and put values in them. We want these values to be there when an object is instantiated. Thus initialization is a good reason for using constructors.

Types of Constructors

Compiler Generated Constructor

If a constructor is not defined by the use the compiler generates it automatically. This constructor has no parameter. It does nothing. Although the compiler will create a default constructor for us, the behavior of the compiler-synthesized constructor is rarely what we want. Thus the default constructor provided by the compiler does no initialization for us.

Simple Constructor

We have earlier discussed that we can write a constructor that takes no argument. The user defined constructor, that takes no argument is called a simple constructor. We know that when a compiler generated default constructor is called, it does no initialization. It does not know whether to put a value in data members like day, month in our previous class **Date**. We can avoid this problem by not writing a class without having its constructor.

A simple constructor can do initialization without any need to take any argument. So we can write a constructor of **Date** class like **Date ();**. When we write such a constructor, it automatically assumes the roll of the default-constructor. The compiler will not call the default constructor. Rather, the constructor written by the programmer will be called whenever an object will be instantiated. It is also a good programming practice to provide a default constructor (i.e. a constructor wit no argument).

Parameterized constructors

We may define a constructor, which takes arguments as well. This constructor will be automatically called when the required number of arguments are passed to it. Through this, we can easily assign the passed values to our class data members for that particular object.

In our previous example of class **Date,** we have written a constructor as follows

Date (int, int, int);

This is a parameterized constructor which takes three arguments of type int.

Constructor Overloading

We can provide more than one constructors by using function overloading. The rules for function overloading are that the name of the function remains the same. However, its argument list may be different. There are two ways to change the

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

argument list. It can either vary in the number or type of the arguments. We cannot have two functions with the same number and type of arguments. In such a case, these will be identical. So it will not be function overloading. The function overloading requires the argument list to be different. The same concept of function overloading applies to constructors. If we supply more than one constructor for a class, it can be called one or the other depending on the way of calling.
As the constructor does not return any thing, so it has no return type. It means that the body of the construct function cannot have any return statement. Otherwise, the compiler will give a syntax error.

Explanation of Constructors

The main purpose of the constructor is to initialize the object in such a manner that it is in a known valid state. Consider the example of **Date** class again. In that example, there were three data members i.e. day, month and year of type int. What values will we give to these data variables by default if we create an object of **Date**? There may be any valid date. We can give a value 01 to day, month and 1901 to year or what ever we want. It will be a known state despite being meaningless. We can write a constructor of class **Date** which takes three arguments int day, int month and int year, and puts values in the data members of the object, being created. Now the question arises when does this happen? It happens when we instanciate an object of class **Date** by writing **Date myDate;** When this line executes in the program, some space for 'myDate' is reserved in the memory. This space contains the space for day, month and year variables. Then control goes to the constructor that assigns values to day, month and year. Being a member of the class, the constructor can write values to the data members of the class that is private. .

In C++ language, we can provide default arguments to functions. As a function, the constructor can take default arguments. Suppose we have written a constructor of class date with the arguments by providing default values to its arguments. We can write a constructor as **Date (int day=1, int month=1, int year=1);**
and create an object of class **Date** as

Date myDate;

This creates an object of type Date. Which constructor will be called? A constructor with no arguments or the parameterized constructor in which each argument has given a value? If we provide a constructor which has default values for all the arguments, it will become the default constructor for the class. Two constructors cannot be considered same. So it will be better not to write a constructor Date (); (constructor with no argument) in case of providing a fully qualified constructor (with default values for all arguments).
Now suppose, we want to initialize the **Date** object properly by passing a character string to its constructor. Is it possible to write such a constructor? Yes, we can write such a constructor. This constructor will take date as a string, say, 01-jan-1999. And in the constructor, we can split up this string with 'string manipulation functions' and assign respective values to day, month and year.

Now we recapture the concept of constructors with special reference to their characteristics.

A constructor is a function which has the same name as the class.
It has no return type, so it contains no return statement.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Whenever an instance of a class comes into scope, the constructor is executed.
The constructors can be overloaded. We can write as many constructors as we require. At one time, the compiler will call the correct version of the constructor".

Utility Functions

The second issue, we usually come across while dealing with the concepts of 'classes and objects' is that a class has a data on one side (normally private part) and functions on the other (normally public part). The functions (methods) are normally written in public part of the class. Are there functions which are private to a class? Answer is yes. The functions of a class may be of two categories. One category contains the member functions which manipulate the data or extract the data and display it. Through these, we can set and get values to manipulate data. These are the functions which are in public interface of the class and manipulate the data in the object. But sometimes, we need such functions that is the requirement of these member functions. Suppose we write a **setDate** function. This function is given an argument and it does the same thing as done by the constructor. In other words, it sets a value of date. Now that function can be public so that it can be called from outside the class. Now we want that the member functions of the class can call this function. But it should not be called from outside. In this case, we put this function in private section of the class. These functions are called utility functions. These are a utility used by other methods of the class. However, they are not functions, supposed to be accessed from outside the class. So they are kept private.

Destructors

The name of the destructor is the same as that of a class with a preceding tilde sign (~). The ~ and name of the class is written as a single word without any space between them. So the name of the destructor of class **Date** will be **~Date**. The destructor can not be overloaded. This means that there will be only one destructor for a class.
A destructor is automatically called when an object is destroyed. When does an object gets destroyed? When we create an object in a function, this is local to that function. When the function exits the life of the object also comes to end. It means that the object is also destroyed. What happens if we declare an object in the main program? When the main program ends, its objects also comes to end and the destructor will be called.
The destructor is normally used for memory manipulation purposes. Suppose we have such a class that when we create an object of it then its constructor has allocated some memory. As we know that we have to free the allocated memory to ensure its utilization for some other program. The destructor is used normally for this purpose to make sure that any allocated memory is de-allocated and returned to free store (heap).

The destructors can be summarized as the following.

The destructors cannot be overloaded.
The destructors take no arguments.
The destructors don't return a value. So they don't have a return type and no return statement in the body.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Now we come to the previously defined class **Date.** Let's see what can we further put in it. We have put in it constructors. We have provided a parameterized constructor without default arguments. So the constructor with no arguments will become default one. We have another constructor with three parameters so that we can pass it the values for day, month and year. There is also provided a destructor. We have written some methods to set day, month and year. These were **setDay, setMonth** and **setYear** respectively. Each one of them takes one parameter, a simple integer. Then we have get functions. The functions **getDay**, **getMonth** and **getYear** return a simple integer. There is also a function **setDate,** which takes three parameters (i.e. day, month and year) and sets them. In set function, we do not simply assign the values to the data members. This can be done through a constructor. Whenever we put data into an object, it is necessary to make it sure that valid values should be stored. For example, if we say **Date myDate ;** and give it values like 35 for day, 13 for month and 2000 for year. The constructor will set these values. But these are invalid values for a date. Here we want that these values should be validated before being assigned to data members. So we write some code for error checking of the values and store only valid values in data members i.e. day, month and year. We do the same thing in set function. Then what is the advantage of using set functions. The set functions are public part of the class and can be called from outside the class and also by the constructor. So write all the code for error checking and to validate the data in set function and call this set function in the constructor. Thus when we create an object of class date, it is written as the following

Date myDate (12,10,2000);

Then an object of **Date** class is created and the constructor of the class that takes three arguments, is executed by passing these three values. In the constructor, we call the set function which sets the values of the data members properly. Thus we get a fine initialization of the data members.

What an Object is ? An object is an instance of a class. When we say an instance that means that this object exists and takes space in the memory. What happens when we create an object i.e. take an instance of the class. A class contains data and methods. Are these methods reproduced for every object? Every object has data of its own as every object is distinct from the other. For example, in case of the date class, there may be objects date1, date2 and date3. These are three different objects having their own value of date. Being distinct objects, they must have distinct space in memory. What about functions inside the class?

Whenever we create an object of a class, the functions of the class take a space in the memory and remain there. There is only one copy of these functions in the memory. The data part of the class takes individual locations in the memory. So if we create three objects of a class, say date, there will be one copy of the functions in the memory at the time of execution of the program. The data will have allocated three spaces in the memory with different values. Now suppose, we want to change the data of date1, there is need of setting month of date1 to 3. So we call *setMonth* function for the object date1. We use dot operator (.) to call the function of an object. We write this as **date1.setMonth(3);** The *setMonth* function is called from the copy of the functions that is in the memory. The object name with dot operator makes sure that the function will operate on the data of that object. Thus only the value of the month of date1 will be set to 3. The values of date2 and date3 will remain untouched. Similarly if we say **date2.setDay(23);** the *setDay* function will be called for object date2 and day of date2 will be set to 23. Thus it is clear that which object calls the function the data of that object is visible to the function and it manipulates only that

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

data. Thus we have not wasted the memory by making separate copy of the functions for each object. All objects of one class share the common functions. On the other hand, every object has its own data space. The overloaded functions and constructors are also found in this single copy and called whenever needed. In the overloaded functions, the appropriate function to be called is resolved by the parameter list (type and number of the arguments to be passed).

In our class **Date,** we need no functionality for the destructor. We write the destructor **~Date** and a cout statement in it. That displays the message 'The object has destroyed' just to demonstrate the execution of the destructor. Similarly we can display a message like 'Date object created' in our constructor function. By this, we can see when the constructor is called. By seeing these messages on the screen we know that the object is being created and destroyed properly. If the constructor function is overloaded, we can put appropriate message in each constructor to know which constructor is called while creating an object. For example in default constructor, we can display a message 'Default constructor is called'.

The following program demonstrates the execution of constructors and destructors. It is the previous example of Date class. It displays appropriate messages according to the constructor called. You will see that the constructor is called depending upon the parameter list provided when the object is being created.

```
/*
A sample program with the Date class. Use of constructors and destructor is shown
here.
A message is displayed to show which one constructor is called
*/

#include <iostream.h>
//#include <stdlib.h>

// defining the Date class
class Date{
   // interface of the class
   public:
      void display();       // to display the date on the screen
      void setDay(int i);   // setting the day
      void setMonth(int i); // setting the month
      void setYear(int i);  // setting the year
      int getDay();         // getting the value of day
      int getMonth();       // getting the value of month
      int getYear();        // getting the value of year

      // Constructors of the class
      Date();
      Date (int, int);
      Date(int, int, int);
      // Destructor of the class
      ~Date ();
   // hidden part of the class
   private:
      int day, month, year;
};
```

vubookshoppk.com

```
// defining the constructor
// default constructor. setting the date to a default date

Date::Date()
{
   day = 1;
   month = 1;
   year = 1900;
   cout << "The default constructor is called" << endl;
}

// Constructors with two arguments

Date::Date(int theDay, int theMonth)
{
        day = theDay;
        month = theMonth;
        year = 2002;
        cout << "The constructor with two arguments is called" << endl ;
}
// Constructors with three arguments

Date::Date(int theDay, int theMonth, int theYear)
{
        day = theDay;
        month = theMonth;
        year = theYear;
        cout << "The constructor with three arguments is called" << endl;
}
//Destructor
Date::~Date()
{
cout << "The object has destroyed" << endl;
}
// The display function of the class date
void Date::display()
{
        cout << "The date is " << getDay() << "-" << getMonth() << "-" << getYear()
<< endl;
}
// setting the value of the day
void Date::setDay(int i)
{
        day = i;
}

// setting the value of the month
void Date::setMonth(int i)
{
```

vubookshoppk.com

```
        month = i;
}

// setting the value of the year
void Date::setYear(int i)
{
        year = i;
}
// getting the value of the day
int Date::getDay()
{
        return day;
}

// getting the value of the month
int Date::getMonth()
{
        return month;
}

// getting the value of the year
int Date::getYear()
{
        return year;
}


/* Main program. We will take three date objects using the three constructors
(default, two arguments and three arguments and display the date.
*/
int main()
{
   Date date1, date2(12,12), date3(25,12,2002);   // taking objects of Date class

   // displaying the dates on the screen
   date1.display();
   date2.display();
   date3.display();
}
```

vubookshoppk.com

Following is the output of the above program.

```
The default constructor is called
The constructor with two arguments is called
The constructor with three arguments is called
The date is 1-1-1900
The date is 12-12-2002
The date is 25-12-2002
The object has destroyed
```

```
The object has destroyed
The object has destroyed
```

vubookshoppk.com

Lecture No. 28

Reading Material

Deitel & Deitel - C++ How to Program ***Chapter 7***
7.6, 7.8

Summary

- Lecture Overview
- Memory Allocation in C
- Memory Allocation in C++
- new Operator and Classes
- Example Program 1
- Classes and Structures in C++
- new Operator and Constructors
- delete Operator and Classes
- Example Program 2
- new, delete outside Constructors and Destructors
- main() Function and Classes
- Class Abstraction
- Messages and Methods
- Classes to Extend the Language
- Tips

vubookshoppk.com

Lecture Overview

In the previous lectures, we have been discussing about Classes, Objects, Constructors and Destructors. In this lecture we will take them further while discussing Memory Allocation.

- We'll see how the memory allocation is done in C++, while discussing memory allocation in C?
- How C++ style is different from the C-style of allocation discussed earlier?
- What are the advantages of C++ approach as compared to that of C?

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Memory Allocation in C

Before further proceeding with the concept of memory, it is better to know what else we can create with classes besides objects.

Recapturing of the concept of 'structures' can help us to move forward. Consider the following statement.

```
struct   abc
{
        int   integer;
        float   floatingpoint;
};
```

We could have declared a structure object as:

```
struct   abc   xyz;        // Declared an object of structure type
```

and access data members inside structure by using dot operator (".") as:

```
xyz.integer   =   2134;
xyz.floatingpoint   =   234.34;
```

Similarly, we could have a pointer to a structure object as:

```
struct   abc*   abcPtr;  // Declared a pointer of a structure type
abcPtr   =   xyz;          // Pointer is pointing to xyz object now
```

We can access the individual data member as:

```
abcPtr->integer   =   2134;
abcPtr->floatingpoint   =   234.34;
```

We can have pointers to different data structures, similarly, pointer to a class object. Here we are going to discuss about Pointers, Classes and Objects.

Let's start by talking about memory allocation. We introduced few functions of memory allocation in C: **malloc()**, **calloc()** and **realloc()**. Using these functions, memory is allocated while the program is running. This means while writing your program or at compile time, you don't need to know the size of the memory required. You can allocate memory at runtime (dynamically) that has many benefits. The classic example will be of an array declared to store a string. If the length of the actual string is lesser than the size of the array, then the part that remains unoccupied will be wasted. Suppose we declare an array of length **6** to contain student name. It is alright if the student name is let's say **Jamil** but what will happen for the student named **Abdul Razzaq**. This is a case where dynamic memory allocation is required.

In C language, the region of memory allocated at runtime is called **heap**. However, in C++, the region of available memory is called **free store**. We have different functions to manipulate memory in both C and C++.

You know that while using **malloc()**, we have to tell the number of bytes required from memory like:

```
malloc(number of bytes required to be allocated);
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Sometimes, we also do a little manipulation while calculating the number of bytes required to be allocated: i.e.

```
malloc( 10 * ( sizeof(int) ) );
```

The **malloc()** returns a void pointer (**void ***). A pointer that points to a void type of memory. So in order to use this memory, we have to cast it to our required type. Suppose, we want to use it for **int**s. For this purpose, you will cast this returned void pointer to **int *** and then assign it to an **int *** before making its further use. The following code is an example of **malloc()** usage.

```
class Date
{
    public:
        Date( ) ;
        Date(int month, int day, int year);
        ~Date ( ) ;
        setMonth( int month ) ;
        setDay( int day ) ;
        setYear( int year ) ;
        int getDay ( ) ;
        int getMonth ( ) ;
        int getYear ( ) ;
        setDate(int day, int month, int year);
    private:
        int month, day, year;
};

    Date *datePtr;                          // Declared a pointer of
Date type.
 int  i;
    datePtr = (Date *) malloc( sizeof( Date ) );    // Used malloc() to
allocate memory
 i = datePtr->getMonth();                   // Returns undefined month value
```

So there is some house-keeping involved during the use of this function. We have to determine the number of bytes required to be allocated and cast the returned void pointer to our required type and then assign it to a variable pointer. Lastly, the memory returned from this function is un-initialized and it may contain garbage.

The contrasting function used to free the allocated memory using **malloc()** is **free()** function. As a programmer, if you have allocated some memory using **malloc()**, it is your responsibility to free it. This responsibility of de-allocation will be there while using C++ functions. But these new functions are far easier to use and more self-explanatory.

vubookshoppk.com

Memory Allocation in C++

The memory allocation in C++ is carried out with the use of an operator called **new**. Notice that new is an operator while the **malloc()** was a function. Let's see the syntax of **new** operator through the following example.

```
new   int;
```

In the above statement, the new operator is allocating memory for an **int** and returns a pointer of **int** type pointing to this region of memory. So this operator not only allocated required memory but also spontaneously returned a pointer of required type without applying a cast.

In our program, we can write it as:

```
int   *iptr;
iptr  =  new   int;
```

So while using **new** operator, we don't need to supply the number of bytes allocated. There is no need to use the **sizeof** operator and cast the pointer to the required type. Everything is done by the **new** operator for us. Similarly, new operator can be used for other data types like **char**, **float** and **double** etc.

The operator to free the allocated memory using **new** operator is **delete**. So whenever, we use **new** to allocate memory, it will be necessary to make use of 'delete' to de-allocate the allocated memory.

```
delete   iptr;
```

The **delete** operator frees the allocated memory that is returned back to free store for usage ahead.

What if we want to allocate space for any array? It is very simple. Following is the syntax:

```
new data_type [number_of_locations];
```

For example, we want to allocate an array of 10 **int**s dynamically. Then the statement will be like this:

```
int   *iptr;
iptr  =  new  int[10];
```

What it does is, it tries to occupy memory space for 10 **int**s in memory. If the memory is occupied successfully, it returns **int *** that is assigned to **iptr**.

Whenever we allocate memory dynamically, it is allocated from free store. Now we will see what happens if the memory in the free store is not sufficient enough to fulfill the request. **malloc()** function returns **NULL** pointer if the memory is not enough. In C++, **0** is returned instead of **NULL** pointer. Therefore, whenever we use **new** to allocate memory, it is good to check the returned value against **0** for failure of the **new** operator.

vubookshoppk.com

Remember, **new** is an operator,it is not a function. Whenever we use **new**, we don't use parenthesis with it, no number of bytes or **sizeof** operator is required and no cast is applied to convert the pointer to the required type.

delete operator is used to free the memory when the allocation is done by using **new** as shown below:

```
int   *iptr;
iptr  =  new   int [10];  // Memory for 10 ints is allocated dynamically.
delete   iptr;            // Allocated is freed and returned to the free store.
```

Can we apply the concept of dynamic memory allocation/deallocation while using new/delete with classes and objects? The answer is obviously yes.

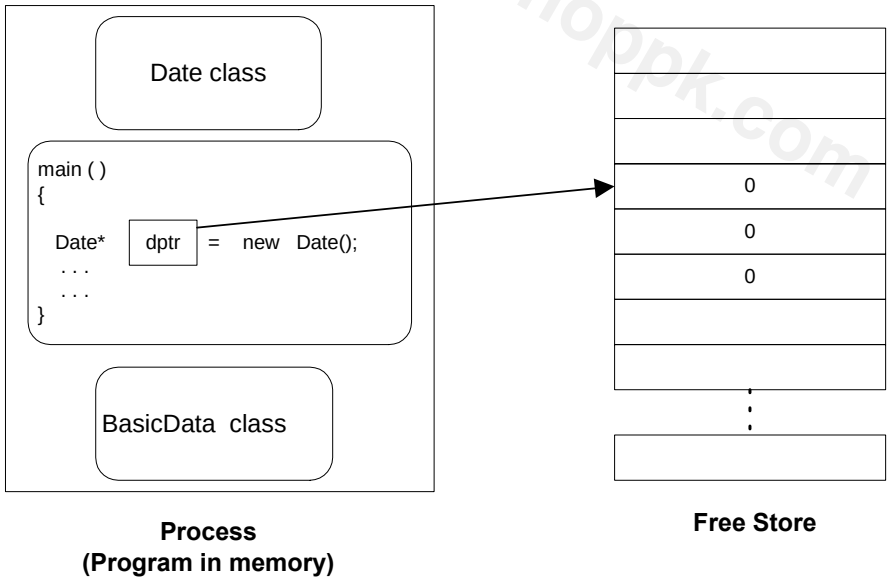
new Operator and Classes

As we declare a pointer to a primitive datatype, similarly, we can have a pointer to a class object.

```
Date   *dptr;    // dptr is a pointer to an object of type Date.
```

Now, we create the object using the **new** operator. Remember, the basic definition of a class i.e. it is a user-defined data type. In other words, the language has been extended to a programmer to have user defined data types. When we use them in our programs, these are used in the same manner as the primitive data types.

```
dptr  =  new  Date;
```



Process (Program in memory)

Free Store

Whatever amount of memory is required for a Date object, is allocated from the free store. A pointer to of Date type is returned back and assigned to the **dptr** pointer variable. Is this all what new is doing? If it is so, can we use **malloc()** function by

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

providing number of bytes required for **Date** object with the help of **sizeof** operator. The answer to this question lies in the further discussion.

```
Date   mydate;
cout   <<   sizeof (mydate);
```

As discussed in the last lecture, whenever we instantiate an object of a class, the data members are allocated for each object. However, the member functions occupy a common region in memory for all objects of a class. Therefore, **sizeof** operator returns the size of the data-members storage excluding the member functions part. In the above statement, the **sizeof** operator returns the sum of the sizes of three integers **day**, **month** and **year**, declared in the **Date** class.

The amount of memory allocated in the above statement using **new** (dptr = new Date;) is same as reflected in the following statement:

```
dptr  =  (Date *)  malloc( sizeof(Date) );
```

The **new** operator in the above statement (dptr = new Date;) has automatically determined the size of the **Date** object and allocated memory before returning a pointer of **Date *** type. Is this all what new is doing? Actually, it is doing more than this. It is also creating an object of type **Date**. C functions like **malloc()** do nothing for object creation. Rather these C functions allocate the required number of bytes and return a **void *** pointing to the allocated memory where the memory might contain garbage. But the **new** operator not only allocates the memory after automatically determining the size of the object but also creates an object before returning a pointer of object's class type.
Additionally, within the call to the **new** operator, the memory assigned to the created object with the use of **new** operator can be initialized with meaningful values instead of garbage (think of C functions like **malloc()**).

How the data members are initialized with meaningful values? Actually, a **constructor** is called whenever an object is created. Inside the constructor, individual data members can be initialized. The C++ compiler generates a default constructor for a class if the programmer does not provide it. But the default constructor does not perform any data members initialization. Therefore, it is good practice that whenever you write a class, use a constructor function to initialize the data members to some meaningful values.

Whenever **new** operator is used to create an object, following actions are performed by it:

- It automatically determines the size of the memory required to store that object, leaving no need for the use of **sizeof** operator.
- Calls the constructor of the Class, where the programmers normally write initialization code.
- Returns pointer of the class type that means no casting is required.

Hence, **new** operator is extremely useful, powerful and a good way of allocating memory.

Let's suppose, we want to allocate space for 10 **int**s as under:

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
int   * iptr;
iptr  =  new  int [10];
```

This **new** statement allocates contiguous space for an array of 10 **int**s and returns back pointer to the first **int**. Can we do this operation for objects of a class? The answer to this question is yes. The syntax in this case will be identical. To create an array of 10 objects of Date type, following code is written:

```
Date   * dptr;
dptr   =   new   Date [10];
int   day   =   dptr->getDay();
```

Here the **new** operator allocates memory for 10 **Date** objects. It calls the default or parameter-less constructors of the **Date** class and returns the pointer to the first object, assigned to the **dptr** variable. Arrow operators (->) is used while accessing functions or data members from the pointer variable.

Example Program 1

```
/* Following program demonstrates the new operator. This program has the problem of
memory leak because delete operator is not called for the allocated memory. */

#include   <iostream.h>

class   MyDate
{
   public:     // public members are below

      /* Parameterless constructor of MyDate class */
      MyDate( )
       {
            cout   <<   "\n Parameterless constructor called ...";
            month   =   day   =   year   =   0;       //  all data member initialized to 0
       }

     /* Parameterized constructor of MyDate class. It assigns the parameter values  to the
……..data members of the class */
      MyDate(int month, int day, int year)
       {
            cout   <<   "\n Constructor with three int parameters called ...";
            this->month   =   month;   // Notice the use of arrow operator ( -> )
            this->day   =   day;
            this->year   =   year;
       }

      /* Destructor of the MyDate class */
      ~MyDate ( )
      {
           cout   <<   "\n Destructor called ...";
```

vubookshoppk.com

```
}

  /* Setter function for the month data member. It assigns the parameter value to
  the month data member */
void setMonth ( int month )
{
     this->month   =   month;
}

 /* Setter function for the day data member. It assigns the parameter value to the
    day data member */
void setDay ( int day )
{
     this->day   =   day;
}

 /* Setter function for the year data member. It assigns the parameter value to the
    year data member */
void setYear ( int year )
{
     this->year   =   year;
}

/* Getter function for the day data member. It returns the value of the day data
   member */
int getDay ( )
{
     return   this->day;
}

/* Getter function for the month data member. It returns the value of the
    month data member */
int getMonth ( )
{
     return   this->month;
}

/* Getter function for the year data member. It returns the value of the year data
   member */
int getYear ( )
{
     return   this->year;
}

/* A function to set all the attributes (data members) of the Date object */
void setDate ( int day, int month, int year )
{
     this->day   =   day;
     this->month   =   month;
     this->year   =   year;
```

vubookshoppk.com

```
    }

  private:              // private members are below
    int  month,  day,  year;
};

main(void)
{
   MyDate  *dptr;                    // Declared a pointer dptr to MyPointer class object
   dptr  =  new  MyDate [10];        // Created 10 objects of MyDate and assigned the
                                     //  pointer to the first object to dptr pointer variable.

// delete should have been called here before the program terminates.
}
```

The output of this example program is as follows:

```
Parameterless constructor called ...
Parameterless constructor called ...
Parameterless constructor called ...
Parameterless constructor called ...
Parameterless constructor called ...
Parameterless constructor called ...
Parameterless constructor called ...
Parameterless constructor called ...
Parameterless constructor called ...
Parameterless constructor called ...
```

vubookshoppk.com

Notice that the constructor is called 10 times with 10 **new** calls but there is no call to destructor. What is the reason? The objects are created with the **new** operator on free store, they will not be destroyed and memory will not be de-allocated unless we call **delete** operator to destroy the objects and de-allocate memory. So memory allocated on free store is not de-allocated in this program and that results in memory leak. There is another point to be noted in this example program, which is not relevant to our topics of discussion today that all the functions are requested to be **inline** automatically as the functions are defined within the class body.

Classes and Structures in C++

Structures and classes in C++ are quite similar. C++ structure is declared with the same keyword **struct** as in C. Unlike C structure, C++ structure can have data and member functions. The difference between class and structure is of visibility. Every data member or function written inside the structure is **public** (visible from outside) by default unless declared otherwise. Similarly, everything declared inside a class is **private** (not visible from outside) by default unless declared as **public**.

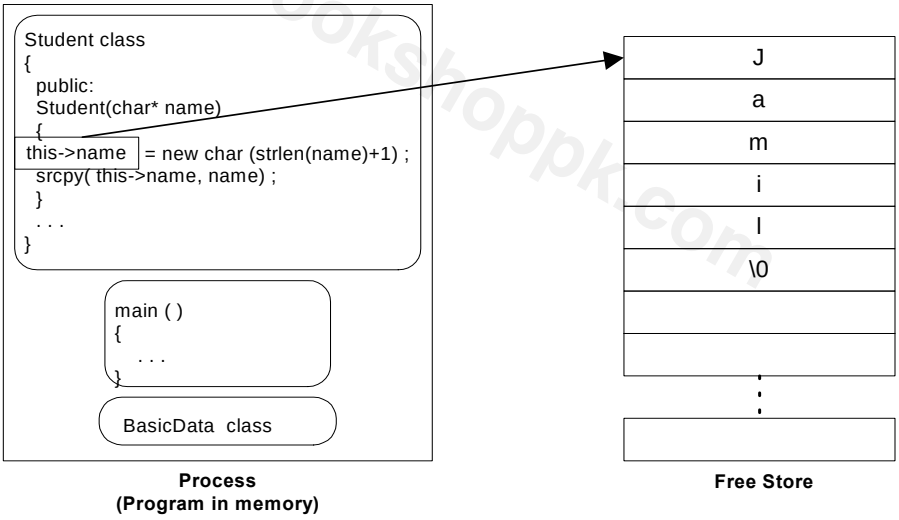
While writing classes, good programming practice is to write **private** keyword explicitly, despite the fact that this is the default behavior. Similarly, while writing structures, it is good to write the **public** keyword explicitly. This averts confusion and increases readability.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Another good practice is to write **public** or **private** keywords only once in the class or structure declaration, though there is no syntactical or logical problem in writing them multiple times.
Also remember while writing a class or a structure that once a keyword is written, say **public,** the declarations falling below this keyword will be **public** until the **private** keyword is mentioned.
There is another keyword protected. We are not using this keyword in this course because that deals with inheritance that is a part of Object Oriented Programming, a separate course.

new Operator and Constructors

It is clear that whenever **new** operator is called to create an object, the constructor is also called for that object. What will happen if we have to call new from inside a constructor function. Can we do that? The answer is definitely yes. There are times when we have to do dynamic memory allocation or create new objects from inside a constructor. For example, we have a **Student** class with attributes i.e. **roll number**, **age**, **height** and **name**. The attributes like **roll number**, **age** and **height** can be contained in **int**s or **float**s but the **name** attribute will require a **string**. Because of the nature of this attribute (as it can have different lengths for different students), it is better to use dynamic memory allocation for this. So we will use **new** operator from within the constructor of **Student** class to allocate memory for the **name** of the



student.

We know whenever we use **new** to allocate memory, it is our responsibility to de-allocate the memory using the **delete** operator. Failing which, a **memory leak** will happen. Remember, the memory allocated from **free store** or **heap** is a system resource and is not returned back to the system (even if the allocating program terminates) unless explicitly freed using **delete** or **free** operators.

Now, we will see how the **delete** works for objects and what is the syntax.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

delete Operator and Classes

As in our **Student** class, as we will be allocating memory from within the constructor of it. Therefore, there is a need to call **delete** to de-allocate memory. What is the appropriate location inside the class **Student** to call **delete** operator to de-allocate memory? In normal circumstances, the location is the destructor of a class (**Student** class's destructor in this case). The destructor is used to de-allocate memory because it is called when the object is no more needed or going to be destroyed from the program's memory. So this is the real usefulness of destructors that these are used to release the system resources including memory occupied by the objects.
As a thumb rule , whenever there is a pointer data member inside our class and pointer is being used by allocating memory at runtime. It is required to provide a destructor for that class to release the allocated memory. A constructor can be overloaded but not a destructor. So there is only one destructor for a class. That one destructor of a class must do house keeping before the object is destroyed. Normal data members **int**, **char**, **float** and **double**, not allocated using **malloc()** or **new** operator, don't need to be de-allocated using **free()** or **delete**. These are automatically destroyed.

Let's be sure that **free()** is used with **malloc()** function while **delete** operator with **new** operator. Normally, **new** will be called in a constructor. However, **delete** will be called in the destructor.

vubookshoppk.com

Example Program 2

```
/* Following program demonstrates the new and delete operators. It deallocates the
memory properly before terminating. */

#include   <iostream.h>


class   MyDate
{
   public:     //public members are below

      /* Parameterless constructor of MyDate class */
      MyDate( )
       {
            cout   <<   "\n Parameterless constructor called ...";
            month   =   day   =   year   =   0;       //  all data member initialized to 0
       }

     /* Parameterized constructor of MyDate class. It assigns the parameter values  to the
……..data members of the class */
      MyDate(int month, int day, int year)
       {
            cout   <<   "\n Constructor with three int parameters called ...";
            this->month   =   month;   // Notice the use of arrow operator ( -> )
            this->day   =   day;
```

```
        this->year   =   year;
 }

/* Destructor of the MyDate class */
~MyDate ( )
{
     cout   <<   "\n Destructor called ...";
}

  /* Setter function for the month data member. It assigns the parameter value to
  the month data member */
void setMonth ( int month )
{
     this->month   =   month;
}

 /* Setter function for the day data member. It assigns the parameter value to the
    day data member */
void setDay ( int day )
{
     this->day   =   day;
}

 /* Setter function for the year data member. It assigns the parameter value to the
    year data member */
void setYear ( int year )
{
     this->year   =   year;
}

/* Getter function for the day data member. It returns the value of the day data
   member */
int getDay ( )
{
     return   this->day;
}

/* Getter function for the month data member. It returns the value of the
    month data member */
int getMonth ( )
{
     return   this->month;
}

/* Getter function for the year data member. It returns the value of the year data
   member */
int getYear ( )
{
     return   this->year;
}
```

vubookshoppk.com

```
    /* A function to set all the attributes (data members) of the Date object */
    void setDate ( int day, int month, int year )
    {
        this->day   =   day;
        this->month   =   month;
        this->year   =   year;
    }

  private:              // private members are below
    int   month,   day,   year;
};

main(void)
{
   MyDate   *dptr;                    // Declared a pointer dptr to MyPointer class object
   dptr   =   new   MyDate [10];      // Created 10 objects of MyDate and assigned the
                                      //  pointer to the first object to dptr pointer variable.

   delete   [] dptr;                  // Deleted (freed) the assigned memory to the objects
}
```

The output of this example program is as follows:

```
Parameterless constructor called ...
Parameterless constructor called ...
Parameterless constructor called ...
Parameterless constructor called ...
Parameterless constructor called ...
Parameterless constructor called ...
Parameterless constructor called ...
Parameterless constructor called ...
Parameterless constructor called ...
Parameterless constructor called ...
Destructor called ...
Destructor called ...
Destructor called ...
Destructor called ...
Destructor called ...
Destructor called ...
Destructor called ...
Destructor called ...
Destructor called ...
Destructor called ...
```

It is very clear from the output that the destructor for all the objects is called to avert any memory leak. The memory allocated using new operator is being de-allocated using the delete operator. Notice the syntax of **delete** while de-allocating an array, the brackets ([]) precedes the name of the array after the **delete** operator.

vubookshoppk.com

new, delete outside Constructors and Destructors

Can **new** be called from some location other than constructor? The answer is yes and we usually need to do that. Suppose, we have an object of **Student** class. The name of the student is: **Abdul Khaliq**. So for the **name** attribute, the space is allocated dynamically to store the string **Abdul Khaliq**. When our program is running and we have already allocated space for the **Abdul Khaliq** string using the **new** operator, after sometime, we are required to increase the size of the string. Let's say we want to change the string to **Abdul Khaliq Khan** now.
So what we can do, without destroying this student object:
De-allocate the **name** previously occupied string using the **delete** operator, determine the size of memory required with the help of **strlen()** function, allocate the memory required for the new string **Abdul Khaliq Khan** using the **new** operator and finally assign the returned pointer to the **name** data member.

Hence, we can call **new** and **delete** operators, not only outside the class to create objects but also within the class. The objects of the same class can have different sizes of memory space like in case of objects of **Student** class, student 1 object can be of one size and student 2 object can be of an another size, primarily varying because of string **name**. But independent of this object size, the destructor of the object remains the same and de-allocates memory for different objects regardless of their different sizes. **delete** operator is used from within the destructor to deallocate the memory. We call **delete** operator to determine the size of the memory required to be de-allocated and only provide it a pointer pointing to it.

Please note that C functions like **malloc()** and **free()** functions can also be used from within C++ code. But while writing classes inside C++ code, we prefer to use **new** and **delete** operators as they are designed to work with classes and objects.

main() Function and Classes

We used to discuss about main() function a lot while writing our programs in C. You might have noticed that while discussing about classes and objects, we are not talking about the **main()** function. This does not mean that **main()** function is not there in C++. It is there but it does not contain as much code in C++ . But as you go along and write your own classess, you will realize that almost 90% of your program's code lies inside the class definitions. So firstly we write our classes and **main()** function is written after classes have been defined. That is why the **main()** function is very small. Our example programs clearly depict this fact.

Class Abstraction

Whenever we write a class, we think about its users. Who are the ones going to use this class? The users are not only the **main()** function of the program but also our colleagues around us. Remember, we only expose interface to our users and not the class implementation. All what users need to know is provided in the interface, the

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

methods signatures and what can be achieved by calling that method. The users do not need to know how the functions or interfaces are implemented, what are the variables, how is the data inside and how is it being manipulated, it is abstract to the users.

Messages and Methods

When we create an object, we ask that object to do something by calling a function. This way of asking objects in Windows operating system is called Messaging or in other words function calling is sending a message to the object. Sending a message is a synonym of calling a method of an object. The word 'method' is from the fact that it is a way of doing something. So the whole program is sending messages and getting responses back. It is a different way of looking at things.

Notice lot of things have been repeated in this lecture many times, the reason is that now, you are required to think differently, more in terms of classes and objects. There are lots of exciting things coming up to be covered later.

Classes to Extend the Language

We know that in C, there is no data type for complex numbers. Therefore, we needed to define our own class for complex numbers. We might use **double** data type for **real** and **imaginary** parts. From basic Mathematics, we also know that whenever two complex numbers are added, real part of one complex number is added into the real part of other complex number and imaginary part of one complex number is added into the imaginary part of other complex number. We might write a function for this operation and might call this as **cadd()**. We might also write other functions for multiplication and division. In C++, the operators like '+', '*' and '/' can be overloaded, therefore, we could **overload** these operators for complex numbers, so that we could easily use these ordinary addition, multiplication, and division operators for complex numbers. Actually, we don't need to write this class on our own because this is already been provided in many C++ libraries.

Remember, there is no primitive data type in C++ for complex numbers but a class has been written as part of the many C++ libraries. Moral of the above paragraph is; by using user defined data types i.e., classes, we can now really extend the language.

Tips

- Classes are one way of extending the C++ language.
- Whenever **new** operator is used, no number of bytes or **sizeof** operator is required and no cast is applied to convert the pointer to the required type.
- Whenever **new** operator is called to create an object, the constructor is also called for that object. It is a good practice that whenever you write a class, use a

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

constructor function to initialize the data members to some meaningful values.

- The usual practice is to use constructor to allocate memory or system resources and destructors to de-allocate or return the resources back to the system.

- In C language, the region of memory allocated at runtime is called **heap**. However, in C++, the region of available memory is called **free store**. There are different functions in C and C++ to manipulate memory at runtime. However, all C functions are useable in C++ code.

- The memory allocated from **free store** or **heap** is a system resource and is not returned back to the system unless explicitly freed using **delete** or **free** operators.

- If the memory in the free store is not sufficient enough to fulfill the request, **malloc()** function returns **NULL** pointer. Similarly, the new function returns **0** in case the request could not be fulfilled.

- Whenever we use **new** operator, the returned value from the **new** should be checked against **0** for any possible failures.

- While writing classes, good programming practice is to write **private** keyword explicitly, despite the fact that this is the default scope. Additionally, the good practice is to write **public** or **private** keywords only once in the class or structure definitions, though there is no syntactical or logical problems in writing them multiple times.

vubookshoppk.com

Lecture No. 29

Reading Material

Deitel & Deitel - C++ How to Program Chapter. 7

7.4

Summary

6) Friend functions
7) Declaration of Friend Functions
8) Sample Program 1
9) Sample Program 2
10) Sample Program 3
11) Friend Classes
12) Summary

vubookshoppk.com

Friend functions

Today, we are going to discuss a very interesting subject i.e. Friend Functions. We will see what is the relationship of friendship with our object-based programming. Before going into details of the subject, it is better to have a fresh look on the definition of 'class'. 'Class is a user defined data type'. The 'class' provides encapsulation facility to the programmer. We can gather data at some place and some function that manipulates that data. In the previous lecture, two keywords, 'private' and 'public' were introduced. We define data members as 'private' that are visible only from inside the class and hidden from the outside. However, 'public data member functions' is the interface of the class available for outside world. Objects are accessed by these functions that can manipulate the private data of the class. We cannot access the private data of the class directly. This concept of data encapsulation and data hiding is very important concept in software engineering. It allows us to separate the interface from the implementation of the class i.e. we can hide how we have done the task and make visible what to do. It is critically important for large and complex systems. Sometimes, a need may arise to access the private data of the class from outside.

Let's talk about the concept of friendship. What you see on the screen during the lecture is the picture of the instructor. This is the public interface. That is all you know. What is inside his mind you never know. It is all 'private'. The instructor has access to his own mind and feelings. But you do not have access to that. Do you know any human being who has access to your mind and feelings? What we call that human

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

being. He is known as friend. Normally other people don't know about our thoughts. Only friends know about it. Friends have access to the inner thoughts and have inner knowledge of a friend. Can we apply this definition to objects?

The friend functions of a class have access to the private data members of class. Despite being a good thing, there is possibility of vulnerability. We are opening our thoughts, inside view for somebody else. Without having 100% trust, it will be risky to make our thoughts and feelings public. We want that our private data is accessible to someone outside, not public for everybody. Otherwise, the data encapsulation and data-hiding concept will be violated. We keep the data members private and declare some specific functions that are not member of the class but friend of the class. As friends, they have access to the inside data structure of the class despite not being members.

Declaration of Friend functions

To declare a friend function, we can put it anywhere in the class. According to the definition of the friend functions, they have access to the private data members of the class. These can also access the private utility functions of the class. The question arises where we should put the friend function whether in the private or public part of the class. Be sure that friend is a very strong statement. It is too strong to be affected by public or private. We can put it anywhere in the class. But remember that friend functions are not member of the class. So their definition will be always outside the class. However, the prototype of the function will be written in the class. We use the keyword 'friend' before the prototype of the function.

```
friend    return_type    friend_function_name(int, char);
```

If we have a class, suppose 'Date' and want to declare a friend function of this class. In the definition of the class, we will write the friend function's prototype with the keyword 'friend'. To access the private data, friend function will need the object. Therefore, usually in the parameter list of friend function, we provide the object of that class. Normally, the programmers work this way. As the friend function is not affected by the private or public keyword, so we can declare it anywhere inside the class definition. Programmers generally declare the friend functions at the top of the class definition. So, the friend functions are declared at the start of the class definition, followed by the private data and public data. This is a guideline. You can develop your own style. We normally make a header file of the class definition and implementation in the other file. The member functions are defined in the implementation file and compiled to get an object file. We declare the friend function in the class definition that is in the header file.

Let's go back to the definition of the friendship. I can declare you my friend and tell you about my inner thoughts and feelings. But it does not work both ways. In other words, friendship is granted, never taken. So, a class can declare a friend function and someone from outside the class cannot declare itself friend of a class. This is also an important concept. If someone from outside the class can declare itself friend of the class, then by definition that external function would have access to the private data member of the class. But this will negate the concept of the encapsulation and data hiding. It does not work this way. A function cannot declare itself friend of a class. Rather, a class has to declare itself that a function is friend of the class or not. So the

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

class declares a friend function. These functions can not declare themselves friend of a class from outside. Once, the friend functions are declared and the class is compiled, no one from outside cannot make his function friend of your class. Outside functions can only view the interface of the class.

Let's summaries this concept. Friend functions are not member functions of the class. The class itself declares the friend functions. The prototype of friend functions is written in the definition of the class with the keyword 'friend'. These functions have access to the private data member of the class, which means they have access to everything in the class. Normally we pass an object of the class to these functions in the argument list so that it can manipulate the data of the object. Style is up to you but normally we write friend functions at the top of the class definition.

Sample Program 1

We have a class with a single private data member of type *int*. We have declared a friend function that accepts an object of that class as argument. We call that friend function *increment*. This friend function will increment the private integer data member of the class. We will give another integer argument to that function that will be added to the data member. The name of the private data member is, for example, *topSecret*. Let's call the class as *myClass*. In the interface, we write *display()* function that will print the value of the *topSecret*. The constructor of the class will initialize the *topSecret* with 100. The definition of the friend function will be outside the class. We do not write the keyword 'friend' with the function definition. It will be a void function, having two arguments as:

```
void increment(myClass *a, int i)
{
        a->topSecret += i;
}
```

Now the increment function has added the value of *i* to the private data member i.e. *topSecret* of the passed object. In the main function, we declare an object of type *myClass* as *myClass x;* On the execution of this statement, an object will be created in the memory. A copy of its data members and functions will also be created besides calling a constructor. The place for *topSecret* will be reserved in the memory while the constructor will assign the value 100 to the variable *topSecret*. Now if we say *x.display();* it will display the value of the *topSecret* i.e.100. After this, we call the increment friend function and pass it *&x* and 10 as arguments. Again we call the display function of *myClass* as *x.display();* Now the value of the *topSecret* will be 110. That means the '*topSecret*' which was the private data member of the class has been changed by the *increment* friend function. Be sure that the increment function is not the member function of the class. It is an ordinary function sitting outside the class but class itself has declared it as friend. So now the friend function has access to the private data member and has the ability to change it. Try to write an ordinary function (not friend function) '*increment2*' which tries to manipulate the *topSecret*. See what will happen? The compiler will give an error that a non- member function can not access the private data of the class.

Here is the complete code of the program.

```
/*
```

vubookshoppk.com

```
A sample program showing the use of friend function,
which access the private data member of the class.
*/

#include <iostream.h>

class myClass
{
        friend void increment(myClass *, int);

        private:
                        int topSecret;

        public:
                        void display() { cout << "\n The value of the topSecret is " <<
topSecret; }
                        myClass();
};
// constructor of the class
myClass::myClass()
{
   topSecret = 100;
}

// Friend function definition
void increment(myClass *a, int i)
{
   a->topSecret += i;    // Modify private data
}

// showing the use of the friend function
void main()
{
        myClass x;
        x.display();
        increment(&x, 10);
        x.display();
}
```

vubookshoppk.com

The output of the program is:

```
The value of the topSecret is 100
The value of the topSecret is 110
```

Sample Program 2

Let's consider some complex example. We have two classes-*myClass1* and *myClass2*. Both classes have one private data member of type *int* i.e. *int topSecret;* Now we want to add the values of private data members of both the classes and display it on the screen. *topSecret* is a private data member of both the classes. One class can not see inside the other class. *myClass1* and *myClass2* are both separate classes. We need a

function sitting outside the classes but can access the private data members of both the classes. Let's call the function as *addBoth*. This function will add the value of *topSecret* of *myClass1* to *topSecret* of *myClass2* and display the result on the screen. We need a function that can look inside both classes i.e. friend of both classes. We know that classes have to declare a function as friend.

The arguments of *addBoth* function will contain *myClass1* and *myClass2*. In the definition of the *myClass1,* we will write the prototype of *addBoth* function as:

```
friend void addBoth(myClass1, myClass2);
```

Can we write this line in the definition of the *myClass1*? We know that if we refer some function as *f(x)* and the function *f()* is not defined or declared before this, the compiler will give an error that function *f()* is not defined. So we at least declare the function before *main()* so that compiler successfully compile the program. So there was declaration of the function before its being called. Now same problem is in our friend function prototype. We are referring both classes in it and our program does not know anything about *myClass2*. We can tackle this problem by writing a line before the definition of the class *myClass1* as:

```
class myClass2;
```

It will declare that *myClass2* is a class having its definition somewhere else. It is the same as we declare functions before *main*. After writing that statement, we can refer *myClass2* in our code. The definition of the class *myClass1* will be as:

```
class myClass1
{
        private:
                int topSecret;

        public:
                void display() { cout << "\nThe value of the topSecret is " <<
  topSecret; }
                myClass1();

        friend void addBoth(myClass1, myClass2);
};

myClass1::myClass1()
{
        topSecret = 100;
}
```

The definition of *myClass2* is also similar to *myClass1*.

```
class myClass2
{
        private:
                int topSecret;
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
        public:
                void display() { cout << "\nThe value of the topSecret is " <<
  topSecret; }
                myClass2();

        friend void addBoth(myClass1, myClass2);
};

myClass2::myClass2()
{
        topSecret = 200;
}
```

You must have noted that we have used the *topSecret* data member in both the classes. Is it legal? Yes it is. There is no problem as one *topSecret* is part of *myClass1* and other is part of *myClass2*. Will there be same problem while declaring the friend function in *myClass2*, i.e. *myClass1* is not known? No. We have already defined the *myClass1*. We have to declare a class only at a time when we are referring to it and it is not defined yet.

In the main program, we will take the object of *myClass1* i.e. *myClass1 a;* The object will be created in the memory and constructor is called to initialize the data members. The value of *topSecret* will be 100. In the next line, we will take the object of *myClass2* as *myClass2 b;* Now *b* is an object of class *myClass2*. The memory will be reserved for it. It has its own data members and the value of *topSecret* will be 200, initialized by the constructor. Now we will display the values of both data members, using *display()* function.

Now we will call the *addBoth(a, b);* As this function is friend of both classes, so it has access to both the classes and their private data members. The definition of *addBoth* function will be as under:

```
        void addBoth(myClass1 a, myClass2 b)
        {
                cout << "\nThe value of topSecret in the myClass1 object is "
<< a.topSecret;
                cout << "\nThe value of topSecret in the myClass2 object is "
<< b.topSecret;
                cout << "\nThe sum of values of topSecret in myClass1 and
myClass2 is " <<                                a.topSecret + b.topSecret;
        }
```

This is an interesting function. Despite not being the member of any class, it can access the data of both the classes. This function is friend of both the classes.

Here is the complete code of the program.

```
/*
A sample program showing the use of friend function,
which access the private data members of two classes.
```

```
*/

#include <iostream.h>

class myClass2; // declaring the class for the friend function in myClass1

// definition of the myClass1
class myClass1
{
   // private data members. Hidden
         private:
          int topSecret;

         // interface of the class
         public:
          void display() { cout << "\nThe value of the topSecret is " << topSecret; }
          myClass1();

         // friend function
         friend void addBoth(myClass1, myClass2);
};
// definition of the constructor.
myClass1::myClass1()
{
         topSecret = 100;
}

// Definition of the myClass2
class myClass2
{
   // private data members. Hidden
         private:
          int topSecret;

         // interface of the class
         public:
          void display() { cout << "\nThe value of the topSecret is " << topSecret; }
          myClass2();

         // friend function
         friend void addBoth(myClass1, myClass2);
};

// definition of the constructor.
myClass2::myClass2()
{
         topSecret = 200;
}

// The definition of the friend function which is adding the topSecret data member of both
```

vubookshoppk.com

```
the classes.

void addBoth(myClass1 a, myClass2 b)
        {
                cout << "\nThe value of topSecret in the myClass1 object is " <<
a.topSecret;
                cout << "\nThe value of topSecret in the myClass2 object is " <<
b.topSecret;
                cout << "\nThe sum of values of topSecret in myClass1 and
myClass2 is " << a.topSecret + b.topSecret;
        }

// main program
void main()
{
        // declaring the objects and displaying the values
        myClass1 a;
        myClass2 b;
        a.display();
        b.display();
        // calling friend function and passing the  objects of both the classes
        addBoth(a, b);
}
```

The output of the program is;

```
The value of the topSecret is 100
The value of the topSecret is 200
The value of topSecret in the myClass1 object is 100
The value of topSecret in the myClass2 object is 200
The sum of values of topSecret in myClass1 and myClass2 is 300
```

The classes have defined and declared this function *addBoth* to be a friend. In each class, we have declared it as a friend function. This function cannot declare itself a friend function for these classes from outside. So be careful about this as a class declares its friend functions. A function out side the class cannot declare itself a friend of the class. The friend functions are not used very often.

Sample Program 3

Now we can expand our previous example. We can define functions *subBoth, mulBoth* and *divBoth* as friend functions of the class, in addition of *addBoth* function. These friend functions can manipulate the data members of the class.

Following is the code of the example that shows the usage of friend functions.

```
/* The following program demonstrate the declaration and uses of friend functions of
a class
We set values in the constructors of the classes. The program prompts the user to
enter a choice of addition, subtraction, multiplication or division. And then performs
the appropriate
```

vubookshoppk.com

```
operation by using the friend functions.
*/

#include <iostream.h>
#include <stdlib.h>

class myClass2;          // declaration of the myClass2 for the friend functions

class myClass1
{
    private:
        float value ;

    public:
        myClass1 ( )
        {
                value = 200 ;
        }

        // friend functions
        friend float addBoth ( myClass1, myClass2 ) ;
        friend float subBoth ( myClass1, myClass2 ) ;
        friend float mulBoth ( myClass1, myClass2 ) ;
        friend float divBoth  ( myClass1, myClass2 ) ;
};

class myClass2
{
    private:
        float value ;

    public:
        myClass2 ( )
        {
           value = 100 ;
        }

        // friend functions
        friend float addBoth ( myClass1 , myClass2 ) ;
        friend float subBoth ( myClass1 , myClass2 ) ;
        friend float mulBoth ( myClass1 , myClass2 ) ;
        friend float divBoth ( myClass1 , myClass2 ) ;
};

void main ( )
{
    myClass1 myClass1Obj ;          //create an object of class myClass1
    myClass2 myClass2Obj ;          //create an object of class myClass2
    char choice;
    cout << "Please enter one of the operator +, -, /, *    " << "followed by Enter " <<
```

vubookshoppk.com

```
endl;
    cin >> choice;

    if ( choice == '+' )
    {
       cout << "The sum is : " << addBoth(myClass1Obj , myClass2Obj) << endl;
    }
    else if ( choice == '-' )
    {
       cout << "The difference is : " << subBoth(myClass1Obj , myClass2Obj) <<
endl;
    }
    else if ( choice == '*' )
    {
       cout << "The multiplication is : " << mulBoth(myClass1Obj , myClass2Obj) <<
endl;
    }
    else if ( choice == '/' )
    {
      cout << "The division is : " << divBoth(myClass1Obj , myClass2Obj) << endl;
    }
    else
    {
       cout << "Enter a valid choice next time. The program is terminating" << endl;
    }

    system ( "PAUSE" ) ;
}

float addBoth ( myClass1 object1 , myClass2 object2 )
{
    return ( object1.value + object2.value ) ;
}

float subBoth ( myClass1 object1 , myClass2 object2 )
{
    return ( object1.value - object2.value ) ;
}

float mulBoth ( myClass1 object1 , myClass2 object2 )
{
    return ( object1.value * object2.value ) ;
}

float divBoth ( myClass1 object1 , myClass2 object2 )
{
    return ( object1.value / object2.value ) ;
}
```

Following is the output of the program.

vubookshoppk.com

```
Please enter one of the operator +, -, /, *  followed by Enter
*
The multiplication is : 20000
```

Friend Classes

We have seen that a class can define friend functions for itself. Similarly a class can be declared as a friend class of the other class. In that case, the function of a class gets complete access to the data members and functions of the other class. So it is an interesting expansion of the definition that not only the functions but also a class can be a friend of the other class. The syntax of declaring a friend class is that within the class definition, we write the keyword *friend* with the name of the class. It is going to be a friend class. i.e. *friend class-name*;
We can also write the word class after the keyword friend and before the class name as

friend class class-name ;

Now let's take another example of a class. Suppose, we have classes *ClassOne* and *OtherClass*. We want to make *OtherClass* a friend class of the *ClassOne***.** So we declare *OtherClass* a friend class in the definition of the *ClassOne* as following.

```
class ClassOne
    {
        friend OtherClass ;
        private:
        //here we write the data members of ClassOne
    };
```

The line

```
friend OtherClass ;
```

can also be written as

```
friend class OtherClass ;
```

The line *friend OtherCalss;* explains that *OtherClass* is a friend of *ClassOne*. If *OtherClass* is the friend of *ClassOne*, all the functions of *OtherClass* will have access to all the inside part of *ClassOne*.
The following code segment shows the declaration of friend class. It shows that *OtherClass* is a friend of *ClassOne* so it has access to the private data of *ClassOne*.

```
class ClassOne
{
        friend class OtherClass;
        private:
        int topSecret;
};

class OtherClass
{
        public:
        void change( ClassOne co )
};
```

vubookshoppk.com

```
void OtherClass::change( ClassOne co )
{
   co.topSecret++;    // Can access private data of class one
}
```

The *friend* keyword provides access in one direction only. This means that while *OtherClass* is a friend of *ClassOne*, the reverse is not true. Here *ClassOne* declares that *OtherClass* is my friend. But it does not work the other way. It does not mean that *ClassOne* has access to the inside data members and methods of *OtherClass*. Thus, it is a one way relationship i.e. the *OtherClass* can look into *ClassOne*, but *ClassOne* cannot look inside *OtherClass*. If we want a two-way relationship, *OtherClass* will have to declare *ClassOne* as a friend class, resulting in a complete two-way relationship.

Like functions, a class cannot declare itself a friend of some other class. A class can declare its friend classes in its declaration and cannot be a friend of other classes by declaring itself their friend. In the above example, *ClassOne* declares that *OtherClass* is my friend class. So *otherClass* can access all the data members and methods (private, public or utility functions) of *ClassOne*. It does not (and cannot) declare that I (*ClassOne*) am a friend class of *OtherClass*. So *ClassOne* has no access to private data members and methods of *OtherClass*. It can access these only if *OtherClass* declares *ClassOne* as its friend. This means that by using the keyword friend, a class gives rights of accessing its data members and methods to other classes and does not get the rights to access other classes.

By declaring friend functions and classes, we negate the concept of data hiding and data encapsulation and show the internal structure of the class to the friends. But the good thing in it is that a class declares its friends while the other functions or classes cannot look inside the class. The disadvantage of friend classes is that if we declare such a relationship of friendship for two classes, this will become a pair of classes. To explain it we go back to the concept of separating the interface and implementation. In case of change in the implementation of *ClassOne*, the private data structure will also change. For example, at first we have an integer variable *int i*; and later, we need two more variables and we write it as *int j, k, l*; As the implementation of *ClassOne* has now changed, the functions of *OtherClass* that wanted to manipulate the members of *ClassOne* will not work now. It is critically important that friend classes should be declared very carefully. When is it necessary? This can be understood by an example from mathematics. We have straight line in math. The equation of straight line is: y = mx + c. Here *m* is the slope of line i.e. the angle which the line makes with x-axis. And *c* is the intercept at y-axis. So if we have to define a straight line, there is need of two numbers i.e. *m* and *c*. Now if we have to define a class *StraightLine*, the private data of it will be *double m, c;* or let's use the names which are self explanatory like *double slope, intercept*; And then in the class, there will be the methods of the class. We can write it as

```
calss StraightLine
{
        //some methods
        private:
                double slope, intercept ;
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
};
```

Now we can also have another class quadratic that also belongs to mathematics. Suppose, we have a parabola, the equation of which is $y= ax^2 + bx + c$. Where a, b and c, for the time being, are real constants. To define this quadratic equation as class, we have to define the three coefficients a, b and c. The statement will be as under:

```
class Quadratic
{
        //some methods
        private:
                double a, b, c ;
};
```

Now we have two classes i.e. StraightLine and Quadratic. In a mathematical problem, when we have given a parabola (a quadratic equation) and a straight line (straight line equation) and are asked to find the point at which the straight line intersects the parabola. To solve it, we setup equations and solve them simultaneously and find out the result, which may be in three forms. Firstly, there is the line that does not intersect the parabola. The second is that it intersects the parabola at one point (i.e. it is a tangential line) and third may be that the line intersects the parabola at two points.

When we setup these equations, we come to know that here the constants *m, c*(of straight line), *a, b* and *c* of quadratic equation are being used. So from a programming perspective if we had an object l1 of type StraighLine and an object q1 of type quadratic. And wanted to find the intersection of l1 with q1. Now here is a situation where we need either a friend function of both classes, so that it can manipulate the data of both classes, or need to declare both classes as friend classes of each other and then write their methods to find the intersection. Similarly we can have many other examples in which a class may need to look into the other class. But it is not some thing to be done all the time. It should be done only when necessary. Use of friend functions is normally a better idea. Using friend classes means that both the classes are linked with each other. If the code in any one of the class is modified i.e. its implementation is changed, we have to recompile both the classes. Due to change in one class, the other class also needs to be changed, necessitating the compilation of both the classes.

So we have lost the principle of separating the interface from the implementation. Now let's talk about the limitations of this friendship business. Firstly, there is no transitive dependency in friend declarations. Suppose I say *student A* is my friend and being a friend he knows my thoughts and ideas. Now the *student A* says "*student B* is my friend" i.e. *student B* knows thoughts and ideas of *student A*. Does it mean that *student B* is also my friend? Does *student B* knows my thoughts and ideas? The answer is no. As I have not declared *student B* a friend of mine, so he (*student B*) does not know about my thoughts and ideas. The same applies to the *friend* definition for classes. The friendship is not transitive. It is not like 'A is a friend of B and B is a friend of C, therefore A is a friend of C'. It does not work. A has to specifically declare 'B is my friend and C is my friend' to make B and C friends of him. There is no transitive dependency in friend declarations.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Secondly, I can declare you to be my friend. This means I have unveiled my thoughts and ideas to you. But I cannot get your thoughts and ideas unless you declare me a friend of yours. So there is no association, which means A saying B is my friend does not imply in any way that A is a friend of B. Here B is a friend of A. But B has to declare 'A' its friend. Thus the *friend* keyword produces one-way relationship.

Summary

The concept of classes allows us to separate implementation from interface.
A class is a user defined data type. In a class, we declare private data members and utility functions so that they cannot be access from outside. Similarly, we declare some parts of the class *public* that become the interface for the class and can be accessed from the outside. These interface methods or public methods can manipulate the data of the class. This is the encapsulation and data hiding.

We have the concept of friend functions. By declaring an external function as a friend function, that function gets the complete access to the inner structure of the class including all private data. When classes need to be interactive, these must be declared friends of each other. Thus we have the concept of friend classes. The use of friend function and class is a useful feature that sometimes we need to use. But we should use it very sparingly and carefully as it basically negates the concepts of encapsulation and data hiding.

The principles of friendship of functions and classes are that the friendship is granted, not taken. So a class declares its friend functions and friend classes. If a class declares another class as a friend, it is not always reciprocal. So declaration and granting of a right is one way. The owner of the right grants it. So the class itself grants the privilege of access to outsider functions or to other classes. It is not transitive. It does not go 'A is a friend of B and B is a friend of C therefore A is a friend of C'. It does not work that way. It is restricted to a one-step relationship. If A is a friend of B, and B is a friend of C. If A wants C to be a friend, it has to declare, "C is my friend".

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Lecture No. 30

Reading Material

Deitel & Deitel - C++ How to Program Chapter. 3

3.17

Summary

13) Reference data type
14) Example 1
15) Difference Between References and Pointers
16) Dangling References
17) Example 2

vubookshoppk.com

Reference data type

Out today's topic is about references. This is a very important topic from the C++ prospective. Today we will see what is a reference, how can we use them. C++ defines a thing by which we can create an alias or synonym of any data type. That synonym is called reference. How do we declare a reference? We declare it by using & operator. Now it is little bit confusing. We have used & as address-of operator and here we are using it for referencing. We will write as

```
int &i;
```

It means that *i* is a reference to an integer. Keep that statement very clear in your mind. It is easier to read from right to left. A reference is a synonym. If we want to give two names to same thing then we use reference. Reference has to be initialized when it is declared. Suppose if we have an integer as *i* and we want to give it second name. We will reference it with *j* as:

```
int &j = i;
```

We declared an integer reference and initialized it. Now *j* is another name for *i*. Does it mean that it creates a new variable? No, its not creating a new variable. Its just a new name for the variable which already exists. So if we try to manipulate *i* and *j* individually, we will came to know that we have been manipulating the same number.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Lets take a look at a very simple example. In the main function we take an int variable *i* and then we write *int &j = i;* Now we assign some value (say 123) to *i*. Now display the value of *i* using *cout*. It will show its value as 123. Display the value of *j* using *cout*. We will not use & operator to display the value of *j*. We will only use it at the time of declaration and later we don't need it. The & is not reference operator rather it acts as reference declarator. The value of *j* will be same as of *i* i.e. 123.

```
int i;
int &j = i;
i = 123;
cout << "\n The value of i = " << i;
cout << "\n The value of j = " << j;
```

Now what will happen if we increment *i* as *i++;* and print the values of *i* and *j*. You will note that the value of *i* and *j* both have been incremented. We have only incremented *i* but *j* is automatically incremented. The reason is that both are referring to the same location in the memory. *j* is just another name for *i*.

What is the benefit of reference and where can we use it? References are synonyms and they are not restricted to int's, we can have reference of any data type. We can also take reference of a class. We wrote a function to show the use of pointers. That function is used to interchange two numbers. If we have two integers *x* and *y*. We want that *x* should contain the value of *y* and *y* should get the value of *x*. One way of doing this is in the main program i.e.

```
int x = 10;
int y = 20;
int tmp;
tmp = y;
y = x;
x = tmp;
```

The values of both *x* and *y* have been interchanged. We can also swap two numbers using a function. Suppose we have a swap function as *swap(int x, int y)* and we write the above code in it, what will happen? Nothing will be changed in the calling program. The reason is call by value. So when the main function calls the function *swap(x, y)*. The values of *x* and *y* will be passed to the swap function. The swap function will get the copies of these variables. The changes made by the swap function have no effect on the original variables. Swap function does interchange the values but that change was local to the swap function. It did not effect anything in the main program. The values of *x* and *y* in the main program remains same.

We said that to execute actual swap function we have to call the function by reference. How we did that. We did not send *x* and *y* rather we sent the addresses of *x* and *y*. We used address operator to get the addresses. In the main function we call swap function as *swap(&x, &y);* In this case we passed the addresses of two integers. The prototype of the swap function is *swap(int*, int*)* which means that swap function is expecting pointers of two integers. Then we swap the values of *i* and *j* using the * notations. It works and in the main program, the values are interchanged. This was a clumsy way. We can use reference in this case with lot of ease. Let us see

vubookshoppk.com

how we can do that. Lets rewrite the swap function using references. The prototype will be as:

```
swap (int &i, int &j);
```

Swap is a function that is expecting *i* as a reference to an integer and the second argument is *j* which is also a reference to an integer. The calling function has to pass references. What will we write in the body of the function? Here comes the elegance of the references. In the body we will treat *i* and *j* as they are ordinary integers. We will take a temporary integer and interchange the values of *i* and *j*.

```
swap (int &i, int &j)
{
        int temp;
        temp = x;
        x = y;
        y = temp;
}
```

In the main program, you will see that the values of two integers have been interchange. What is the way to call this function? In the main program, we will call this function as *swap(x, y)*. Its an ordinary call but the function is expecting addresses which is automatically done. The memory locations of the integers are passed and the function is interchanging the original numbers. This is one beautiful example in which we avoided all the cumbersome of pointer notation. What is the downfall of this? As nothing comes for free. In this case when you are reading the main function you will see *swap (x, y)* which seems a call by value. This is a rule of C/C++ that when we pass two variables to some function they are passed by values. You will have to look for the definition of swap function to realize that it is not call by value but is call by reference. Second thing is if we have another swap function, which is receiving two integers. Can we define two functions as *swap(int x, int y)* and *swap(int &x, int &y)*? One function is receiving two integers and other is receiving two references of integers. Can we do that? Types are different so we can overload. Unfortunately not, in the main function the way to call both functions is same i.e. *swap(x, y)*. How does the compiler know that which functions is being called? There is no way for the compiler to find out. Therefore there is an ambiguity and that is not allowed. The only thing to realize is the side effect. Side effects are critical to take care of whenever you are doing call by reference. Here in this example we do want that two numbers should be interchanged. There may be some situation where we want to send the references and don't want that original data should be affected. These situations arise when we want to pass a large data structure to a function. To understand this we have to understand how the function call executes. We have discussed it before, now lets recap it. In real world, suppose I am reading a book. While reading I notice a word which I think I have to look up. I stop reading, markup the page and then look that word in dictionary or encyclopedia. While reading the definition of that word I look another special word which I need to lookup. I put a marker here and go for looking the definition of that new word. Eventually I understand all the words I need to look up. Now I want to go to the point at which I left the study. I close the dictionary or encyclopedia and goes back to the original book which I was studying. Now I understand all the words in my study and continue the study from the point where I

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

left. If we think about it, it was a function call. We were doing some work, suddenly we call a function and stop that work and execution went to the function. When the execution of the function came to end, we came back to our calling function and continued with it. Computers do the same work with stack. So when the program comes back from the function it should know the point at which it lefts. We supposed here a word to look up, now consider that it was a paragraph or essay I am going to look up. Now to lookup that essay in other books I have to take the entire paragraph or essay with me to that book. Think about the stack. On the stack, the original condition of the program (state) has saved. Now we put our essay or paragraph on it and then opened the other book and searched the book for this essay. In this way, we want to explain that the thing we passed to the function from the main was itself a huge/large thing (as we resemble it with paragraph or essay). So there was a big overhead in writing that thing out into a temporary space in memory and then picking it up and looking it up.

We can make this process more efficient. The issue is that in this example we do not want to change the paragraph or essay which we are going to look up. We only want to look it up. We want only to use it but don't want to change its words. Unfortunately the baggage that comes with doing this is that first make a copy of this (essay) then go with this copy and when the work with it ends, leave (through away) the copy and start the original work. This is inefficient.

But if we took the reference of that essay and passed the address of it and went to the function to look it up. There is a danger that comes with the address, that is while looking up that essay I underlined different words and when I came back to original book I saw that these line marks were also there. Thus we passed something by value rather we passed something by reference. By passing the reference, we actually pass the original. Think about it in another way. We go to a library and said the librarian to issue us a book, which we want to take home for study. Suppose, that book is the only one copy available in the library (or in the world). The librarian will not issue the book. Because it is the only copy available in the world. He does not want to issue this original book to someone as someone can marks different lines with a pen and thus can damage the original book. The librarian will do that he will take a photocopy of that book and issue it. Making a photocopy of the book and then take the book is a bothersome work.

Here we don't want to damage the book. We just want to read it. But can I somehow take the original book? Put it in a special polythene bag and give it to you in such a way that you can read it without damaging it. By doing this we get efficiency but danger is still there. This is actually a call by reference. We have the reference (the original book). If we do something to the original book, the library book will be damaged. Can we somehow prevent this from happening? And also have the efficiency of not having to make a copy.

Now come back to the computer world. Suppose we have a data structure. There is a string of 1000 characters in it. We want to pass that data structure to a function. If we pass it by value which is sake, the original structure will not be affected. We first will copy that string of 1000 characters at some place, which is normally made on the stack. Then the function will be called. The function will take the copy of these 1000 characters and will manipulate it. Then it will give the control back to the caller

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

program and will destroy that copy of string. For efficiency, we want that instead of making a copy of this string, its reference should be written. We have been doing this with pointers and addresses. So we write there the address and pass it to the function. How we can prevent the side effects? There may be these side effects with references. So be very careful while using references with function calls.

Can we do something to prevent any changes? The way we do it is by using the *const* key word. When we write the *const* key word with the reference, it means that it is a reference to some thing but we cannot change it. Now we have an elegant mechanism. We can get the efficiency of call by reference instead of placing a string of 1000 characters on the stack, we just put the address of the string i.e. reference on the stack. In the prototype of the function, it is mentioned that it takes a *const*. This is a reference that may be to a char, int, double or whatever but it is a *const*. The function cannot change it. The function gets the address, does its work with it but cannot change the original value. Thus, we can have an efficiency of a call by reference and a safety of a call by value. To implement all this we could have used the key word *const* with an address operator or a pointer but we can use a reference that is an elegant way. There is no need in the function to dereference a reference by using * etc, they are used as ordinary variable names.

Example 1

Now let us have an example. Here we defined a structure *bigone* that has a string of 1000 characters. Now we want to call a function by three different ways to manipulate this string. The first way is the call by value, which is a default mechanism, second is the call by reference using pointers and the third way is call by reference using reference variables. We declared the prototypes of these functions. Here we declared three functions. The first function is *valfunc* which uses a call by value. We simply wrote the value of the structure. The function prototype is as under.

```
void valfunc( bigone v1 );
```

The second function is *ptrfunc* in which we used call by reference using pointers. We passed a pointer to the structure to this function. The prototype of it is as follows.

```
void ptrfunc( const bigone *p1 );
```

The third function is *reffunc* which uses the way of calling by reference using references. We wrote its prototype as

```
void reffunc( const bigone &r1 );
```

Note that we wrote & sign with the name of the variable in the prototype of the function, we will not write it in the call of the function.
In the main program, we called these function. The call to the *valfunc* is a simple one we just passed the name of the object of the structure i.e. *v1*. As the function is called by using the call by value the manipulation in the function will not affect the original value. We wrote it as:

```
valfunc ( bo );
```

In this call a copy of *bo* is placed on the stack and the function uses that copy.

Next we called the function *ptrfunc*. We passed the address of the structure to *ptrfunc* by using the & operator. Here we are talking about the function call (not function prototype) and in function call we write *ptrfunc (&bo) ;* which means we passed the

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

address of *bo* (the object of structure) to the function. The efficiency here is that it writes only the address of the object to the stack instead of writing the whole object.

The call to the third function *reffunc* is simple and looks like the call by value. There is no operator used in this call it is simply written as:

reffunc (bo) ;

Here we cannot overload the valfunc and reffunc, their names must be different. Otherwise the calls look same and become ambiguous.

The pointer call and reference call are sending the references to the original structures so these are dangerous. If we want to prevent the function from changing that then we should define the function by *const* keyword with its argument pointer or reference. Then the function can not modify the original value, it can only read it. So by this we get the efficiency of the call by reference and the safety of the call by value.

The complete code of the example is given here.

```
// Reference parameters for reducing overhead
//   and eliminating pointer notation

#include <iostream.h>

// A big structure
struct bigone
{
  int serno;
  char text[1000];   // A lot of chars
} bo = {123, "This is a BIG structure"};

// Three functions that have the structure as a parameter
void valfunc( bigone v1 );                    // Call by value
void ptrfunc( const bigone *p1 );      // Call by pointer
void reffunc( const bigone &r1 );      // Call by reference

// main program
void main()
{
  valfunc( bo );                 // Passing the variable itself
  ptrfunc( &bo );                // Passing the address of the variable
  reffunc( bo );                 // Passing a reference to the variable
 }

//Function definitions
// Pass by value
void valfunc( bigone v1 )
{
  cout << '\n' << v1.serno;
  cout << '\n' << v1.text;
}
// Pass by pointer
void ptrfunc( const bigone *p1 )
```

vubookshoppk.com

```
{
  cout << '\n' << p1->serno;     // Pointer notation
  cout << '\n' << p1->text;
}
// Pass by reference
void reffunc( const bigone &r1 )
{
  cout << '\n' << r1.serno;       // Reference notation
  cout << '\n' << r1.text;
}
```

Following is the output of the above program.

```
123
This is a BIG structure
123
This is a BIG structure
123
This is a BIG structure
```

Difference Between References and Pointers

The reference in a way keeps the address of the data entity. But it is not really an address it is a synonym, it is a different name for the entity. We have to initialize the reference when we declare it. It has to point to some existing data type or data value. In other words, a reference cannot be NULL. So immediately, when we define a reference, we have to declare it. This rule does not apply to functions. When we are writing the argument list of a function and say that it will get a reference argument, here it is not needed to initialize the reference. This reference will be passed by the calling function. But in the main program if we declare a reference then we have to initialize it. When a reference is initialized, we cannot reassign any other value to it. For example, we have *ref* that is a reference to an integer. In the program we write the line *int &ref = j ;*

Here *j* is an integer which has already been declared. So we have declared a reference and initialized it immediately. Suppose we have an other integer *k*. We cannot write in the program ahead as *ref = k;* Once a reference has defined, it always will refer to the same integer location as *j*. So it will always be pointing to the same memory location. We can prove this by printing out the address of the integer variable and the address of the reference that points to it.

In programming, normally we do not have a need to create a reference variable to point to another data member or data variable that exists, because creating synonym that means two names for the same thing, in a way is confusing. We don't want that somewhere in the program we are using *i* (actual name of variable) and somewhere *ref* (reference variable) for manipulating the same data variable. The main usage of it is to implement the call by reference through an elegant and clean interface. So reference variables are mostly used in function calls.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

The difference between pointers and references is that we can do arithmetic with pointers. We can increment, decrement and reassign a pointer. This cannot be done with references. We cannot increment, decrement or reassign references.

References as Return Values

A function itself can return a reference. The syntax of declaration of such a function will be as under.

datatype& function_name (parameter list)

Suppose we have a function *myfunc* that returns the reference to an integer. The declaration of it will be as:

int & myfunc() ;

Dangling Reference

The functions that return reference have danger with it. The danger is that when we return a value from such a function, that value will be reference to some memory location. Suppose that memory location was a local variable in the function which means we declare a variable like *int x;* in the function and then returns its reference. Now when the function returns, *x* dies (i.e. goes out of scope). It does not exist outside the function. But we have sent the reference of that dead variable to the calling function. In other words, the calling program now has a reference variable that points to nowhere, as the thing (data variable) to which it points does not exist. This is called a dangling reference. So be careful while using a function that returns a reference. To prevent dangling reference the functions returning reference should be used with global variables. The function will return a reference to the global variable that exists throughout the program and thus there will be no danger of dangling reference. It can be used with static variables too. Once the static variables are created, they exist for the life of the program. They do not die. So returning their reference is all right.

So, never return a reference to a local variable otherwise, there will be a dangling reference. Some compilers will catch it but the most will not. The reason is that the function that is returning a reference has defined separately. It does not know whether the reference is to a global or local variable, because we can do many manipulations in it and then return it. But normally compilers will catch this type of error.

Example 2

Let us look at an example of functions returning references. First, we declare a global variable that is an integer called *myNum* and say it is zero. Then we declare a function *num* that returns a reference to an integer. This function returns *myNum*, the global variable, in the form of reference. So now when there will be a function call, the return of the function will be a reference to the global variable called *myNum*. Now we can write the main function. Here we write *myNum = 100* ; This assigns a value 100 to the global variable. Next we write

```
int i ;
i = num () ;
```

Now a reference to *myNum* is returned. We would want to assign a reference to a reference but we can use it as an ordinary variable. Thus that value is assigned to *i*.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Now look at the next line which says *num () = 200* ; We know that the left hand side of the assignment operator can only be a simple variable name, what we called l-value (left hand side value). It cannot be an expression, or a function call. But here in our program the function call is on left hand side of the assignment. Is it valid? In this case it is valid, because this function called *num* is returning a reference to a global variable. If it returns a reference, it means it is a synonym. It is like writing *myNum = 200* ; The example shows that it can be done but it is confusing and is a bad idea. We can put a reference returning function on the left hand side of an assignment statement but it is confusing and bad idea.

Following is the code of the example.

```
/*Besides passing parameters to a function, references can also be used to return values from a function */

#include <iostream.h>

int myNum = 0;    // Global variable

int& num()
{
   return myNum;
}
void main()
{
   int i;
   i = num();
   cout << " The value of i = " << i << endl;
   cout << " The value of myNum = " << myNum << endl;
   num() = 200;   // mynum set to 200
   cout << " After assignment the value of myNum = " << myNum << endl;
}
```

vubookshoppk.com

Following is the output of the program.

```
The value of myNum = 0
 After assignment the value of myNum = 200
```

The references are useful in implementing a call by reference in an efficient fashion and writing the function very elegantly without using dereference operators.

We use & sign for declaring a reference. In the program code, how do we find out that it is a reference or an address is being taken? The simple rule is that if in the declaration line there is reference symbol (& sign) with the variable name then that is a reference declaration. These will be like *int &i, float &f* and *char &c* etc. In the code whenever we have simply &i, it means we are taking address. So it's a simple rule that when, in the code, we see a data type followed by & sign, it's a reference. And when the & sign is being used in the code with a variable name then it is the address of the variable.

In C and C++ every statement itself returns a value. It means a statement itself is a value. Normally the value is the value of left hand side. So when we write *a = b*; the

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

value of *b* is assigned to *a* and the value of *a* becomes the value of the entire statement. Therefore when we write *a = b = c* ; first *b = c* executes and the value of *c* is assigned to *b*. Since *b = c* is a statement and this statement has the value of *b*. Now *a* takes the value of this statement (which happened to be *b*). So *a = b* also works. Similarly *a + b + c* also works in the same way that the value of *c* is added to *b* and then this result is added to *a*.

What happens when we write *cout << "The value of integer is " << i << endl* ;
Here first extreme right part will be executed and then the next one and so on or the other way. On the screen the "The value of integer is" displayed first and then the value of the *i* and in the end new line. So it is moving from left to right. When *cout* gets the first part i.e. "The value of integer is", this is a C statement. When this will be executed, the sentence "The value of integer is" is displayed on the screen. But what will be its value? That has to do something with the next << part and is needed with this << sign. We know that we need *cout* on the left side of << sign. So actually what happened is when the first part of the statement is executed. When the statement *cout << " The value of integer is"* executed *cout* is returned. The next part is *<< i* and it becomes *cout << i;* the value of *i* is printed and as a result of the statement *cout* is returned again which encounters with *<< endl;* and a new line is inserted on the screen and *cout* is returned as a result of the statement execution. The return of the complete statement remains *cout*. The *cout* is stream, it does not have value per se. The reference to the stream is returned. The same reference which we have discussed today. The same thing applies to operators like +, -, *, /. This will also apply to = (assignment operator) and so on. We will be using lot of reference variables there.

Summary

We have learned a new data type i.e. reference data type. We said that reference is synonym or alias for another type of data. Take int's synonym or double's synonym. In other words, it's the second name of a variable. Then we talk about some do's and dont's. Normally we do not use two names for the same variable. It's a bad idea and leads to confusing the programmer. Then we found the most useful part of using a reference. If we have to implement call by reference with function then using the prototype of the function which is expecting references and it leads to clean programming. You use the names of the arguments without using any dereferencing operator like *. The most useful part is implementing the call by reference. Then we looked at the difference of pointers and references. We cannot increment the reference variable. Arithmetic is not allowed with references but most importantly, reference variables must be initialized when they are declared. This is import. We can declare pointers and later can assign it some value. The use of reference with classes will be covered later. We have also seen a preview of the usage of references. In that preview we have learned new things that every statement itself has some value and that value is returned. Use it or not it's a different issue. We call a function on a single line like *f(x);* may be *f(x)* returns some value and we did not use it. Not a problem. Similarly if we say *a = b;* this statement itself have some value whether we use it or not. Then we see how the cout statement is executed. Every part of the statement returns some value which is the reference to *cout* itself. It becomes the reference to the stream.
How these references will be declared and used? We will cover this with operator overloading. Try to write some programs using references and implement call by reference using references instead of pointers.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Tips

- The use of reference data type is the implementation of call by reference in an elegant way.
- We cannot do arithmetic with references like pointers.
- Reference variables must be initialized immediately when they are declared.
- To avoid dangling reference, don't return the reference of a local variable from a function.
- In functions that return reference, use global or static variables.
- The reference data types are used as ordinary variables without any dereference operator.

Lecture No. 31

vubookshoppk.com

Reading Material

Deitel & Deitel - C++ How to Program ***Chapter 8***

8.2, 8.3, 8.4, 8.6, 8.7

Summary

- Lecture Overview
- What is Operator Overloading and Why is it Required
- Where is it Relevant to Apply
- Operators to Overload
- Restrictions on Operator Overloading
- Examples of Operator Overloading
- Non-member Operator Functions
- Example Program 1
- Example Program 2
- Tips

Lecture Overview

The topic of this lecture is **Operator Overloading**. In previous lectures, we discussed about it a bit while discussing about references. So we will see in detail what is operator overloading, how to overload operators, where it is relevant to apply and what are the restrictions on it.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

What is Operator Overloading and Why is it Required?

Operator overloading is to allow the same operator to be bound to more than one implementation, depending on the types of the operands.

As you know that there are standard arithmetic operators in C/C++ for addition (+), subtraction (-), multiplication (*) and division (/). We should only use these operators for their specific purposes. If we want to add two **int**s, say **i** and **j**, the addition will take place in the following manner i.e. **i + j**. To add two **double** numbers, we use the same operator and write **d1 + d2**. We may add two floats with the help of the same operator as **f1 + f2**. Similarly other operations of **-**, ***** and **/** on the primitive types (sometimes called as native or built-in types) can be employed. In other words, these operators are already overloaded for primitive types in C++. But these C++ operators cannot be used for classes and their objects. We have to write our own operator functions that can work with objects.

Let's take an example of complex numbers. There are two parts of a complex number i.e. **real** and **imaginary**. As complex numbers are part of mathematical vocabulary, so the mathematical manipulations are done on them like addition, subtraction and multiplication. Suppose, we write our own class for complex numbers named **Complex**, but we can't add two complex numbers **c1** and **c2** as **c1 + c2** because until now we don't know how to write it. Although, we are able to write a function say **cadd()** to serve this purpose.

```
Complex   cadd ( Complex c1,  Complex  c2 ) ;
```

It accepts two complex numbers as parameters and returns back the resultant complex number. But the usage of this function to add two complex numbers is generally clumsy. It gets more cumbersome and complex if we want to carry out cascading operations like

i + j + k . It is better to use the standard operators of +, -, ***** and / as they are more readable and elegant.

Where is it Relevant to Apply?

Firstly, the operator overloading gets relevant whenever there is the application of the mathematical functions of addition, subtraction, multiplication and division. Complex number is one example of it. As discussed earlier, in case of **Date** class, the operators can be effectively used to get the future or past dates.

Secondly, the operators are also used sometimes in case of non-mathematical manipulation. The example of **String** class to manipulate strings help us understand it in a better way. The operator + can be used to concatenate two strings. Previously, we used **strcat()** function declared inside **string.h** header file to concatenate two strings. As compared to **strcat()**, the use of + to concatenate two strings is definitely easier and more readable. But there is a little bit cost associated with this process of operators overloading.

The cost is involved whenever we overload an operator. We have to write a function and make use of the operator semantics correctly while implementing the function.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

This means that the function written to overload + operator should do addition or concatenation of strings in case of String objects.

Operators to Overload

There are two types of operators to overload:

1. Unary
2. Binary

Unary operators are the ones that require only one operator to work. Unary operators are applied to the left of the operand. For example, ^, **&**, ~ and **!**.

Binary operators require two operands on both sides of the operator. +, **-**, *****, /, **%**, =, **<** and **>** are examples of binary operators.

The complete list of C++ operators that can be overloaded is as follows:

+	-	*	/	%	^	&
|	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	|=
<<	>>	>>=	<<=	==	!=	<=
>=	&&	||	++	- -	-> *	,
[ ]	( )	new	new[ ]	delete	delete[ ]	

The following operators can't be overloaded.

.	:	::	.*	?	sizeof

Let's start with operator overloading mechanism. Consider an object **date** of the **Date** class. The data member **day** can be accessed as follows:

date.day = 2;

In this statement, the **day** data member of the **date** object is accessed and assigned value **2**. This expression (**date.day**) is driven by the object name at left.
Similarly, while using operators, the statement like **a + b** is driven by the object at the left. In this case, + operator function for the object **a** will be called and **b** object is passed explicitly to the + operator function as an argument. The rules of function overloading are applied to the operator overloading. We cannot write two + operator functions with exactly identical parameters. Following the overloading rules, the two operator functions have to be different by the type or number of arguments.

The syntax of the prototype of the overloaded operator function is:

return-type operator *operator-symbol* *(parameter-list);*

operator is the keyword here. An example of this will be as follows:

Complex operator + (Complex &);

vubookshoppk.com

We sometimes write only operator to refer to the operator function in our discussion.

Restrictions on Operator Overloading

There are some restrictions on operator overloading.

- The operator overloading functions for overloading **()**, **[]**, **->** or the assignment (=) Operators must be declared as class members.
- The arity (number of operands) cannot be changed. If you are overloading an operator that requires two operands e.g. *****. It cannot be used as a unary operator that requires one operand.
- No new operators can be created. Like in Fortran language, we have ****** as 'raise to the power (exponent) operator' but this operator does not exist in C++. Therefore, it can't be overloaded. Hence, only existing operators of C++ are used.
- Overloading can't be performed for the built-in (sometimes called primitive or native) data types. For example, we cannot change how two **int**s are added. That means that operators are overloaded to use with defined data types like classes.
- Precedence of an operator cannot be changed. For example, the ***** has higher precedence than **+.** This precedence cannot be changed.
- Associativity of an operator cannot be changed. If some operator is right associative, it cannot be changed to be left associative.

vubookshoppk.com

Examples of Operator Overloading

Let's take the complex number's class **Complex** and define a + operator function.

We know that when we write the following line:

```
x = y + z ;
```

y and **z** operands are take part in the addition operation but there is no change in them due to this operation. This is the + operator's functionality. The resultant is being assigned to the variable **x**. This is assignment operator's functionality.

Now we will discuss a little bit about the assignment operator as well. Let's say we write the following statement for two complex numbers **c1** and **c2**.

```
c1 = c2 ;
```

Here **c2** is being assigned to **c1**. Will this assignment work when we have not written any assignment operator function for complex number? Apparently, it looks that the statement will produce a compilation error (as there is assignment operator defined by us) but this is not true. Whenever, we write our own class and compile it, the compiler automatically generates a default assignment operator. The default assignment operator makes a member to member assignment. This works fine unless there is a pointer data member inside our class and that pointer is pointing to some data inside memory. For that case (when there is a pointer data member) we have to write our own assignment operator otherwise the default assignment operator works fine for us. That will be discussed in the subsequent lectures.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

By definition of addition of complex numbers, we know that whenever two complex numbers are added, the real part of one number is added into the real part of other number. Similarly, the imaginary part of one number is added to the imaginary part of the other number. We also know that when a complex number is added to another complex number, the resultant is also a complex number consisting of real and imaginary parts. This addition of real, imaginary parts and return of resultant complex number is the functionality of the + operator function we are going to write.
Another thing to decide for this + operator is whether this operator will be a member operator or a **friend** operator. Normally, operators are member operators but there are situations when they cannot be member operators. In case of member operator, following is the syntax of its prototype:

```
Complex operator + (parameter-list);
```

For member operator, the object on the left side of the + operator is driving this + operation. Therefore, the driving object on the left is available by **this** pointer to + operator function. But the object on the right is passed explicitly to the + operator as an argument.
We can define a member operator as under:

```
1.  Complex Complex :: operator + (Complex c)
2.  {
3.      Complex temp ;
4.      temp.real = real + c.real ;
5.      temp.imag = imag + c.imag ;
6.      return temp ;
7.  }
```

Let's see this code line by line.

Line 1 indicates that the return type is **Complex**, it is an **operator +** function and it is accepting a **Complex** object by value as an argument.
In line 3, a local **Complex** object is declared, called **temp**.
In line 4, **real** part of the calling object (that is the one, driving) on the left of the + operator is being added to the **real** part of the object **c**, where **c** is passed as an argument.
In line 5, **imag** part of the calling object (that is the one, driving) on the left of the + operator is being added to the **imag** part of the object **c**, where **c** is passed as an argument.
In line 6, the **Complex** object **temp** containing the resultant of + operation is being returned by value.

In our code, we can write something as:

```
Complex c1, c2, c3 ;
. . .
. . .
c3 = c1 + c2 ;
```

In the above statement (c3 = c1 + c2;), c1 is the object that is calling or driving the + operator. **c2** object is being passed as an argument to the + operator. So c1 and c2 objects are added by the + operator and resultant **Complex** object containing the

vubookshoppk.com

addition of these two numbers is returned back. That returned **Complex** object is assigned to the **c3 Complex** object using the default assignment operator (that is created by the C++ compiler automatically).

What happens if we want to add a **double** number to a complex number (a instance of **Complex**)? Like the following:

```
c3 = c1 + d ;
```

This + operation is driven by the **c1** object of **Complex** while double number **d** of type **double** is passed as argument. Therefore, our above written + operator is not useable for this operation of addition. We need to overload + operator for accepting a parameter of type **double**, i.e. we need to write another operator function. The definition of this newly overloaded + operator is:

```
Complex Complex :: operator + (double d)
{
        Complex temp ;
        temp.real = real + d ;      // d is added into the real part
        temp.imag = imag ;
        return temp ;
}
```

By now, you should have noticed that operator overloading and function overloading are quite similar.

When we write the following statement:

```
c3 = d + c1;
```

The operand on the left of + operator is a **double** number **d.** Therefore, this + operation should be driven by (called by) the **double** number. Until now, we have not written such an operator. Our previously written two + operators were driven by the **Complex** object. Operator functions, not driven by the class type objects, are kept as **friend**s to the class. **friend** is the keyword used to declare such functions. A **friend** function to a class also has access to the private members of that class.

```
friend Complex operator + (double d, Complex c)
{
        Complex temp;
        temp.real = d + c.real;    // d is added into the real part of c
        temp.imag = c.imag;
        return temp;
}
```

You might have noticed that all the three overloaded + operator functions are accepting and returning variables by value. To make these functions better, we can also use references. So our first member + operator's prototype can be rewritten as:

```
Complex& operator + (Complex& c);
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Now this operator function is accepting a complex number **Complex** by reference and returning a reference to the resultant complex number.

As discussed above, in case of assignment, the default assignment operator is used because we have not implemented (overloaded) our own assignment operator ('='). But in case, we want to perform the following operation where the two operands are added and the resultant is assigned to one of them as:

```
c1 = c1 + c2;
```

There is one operator (+=) that can be used to do both the operations of addition and assignment instead of doing these operations separately within **operator +** and **operator =.** So we can overload this one operator (+=) here to make the code more efficient and reduce our work. Therefore, instead of writing:

```
c1 = c1 + c2;
```

We will write:

```
c1 += c2;
```

We will write our **operator +=** as:

```
void Complex :: operator += ( Complex& c )
{
        real += c.real;
        imag += c.imag;

}
```

vubookshoppk.com

Non-member Operator Functions

Now we are much clear that when an operator function is implemented as a member function, the leftmost operator must be a class object or reference to a class object of the operator's class.

When an operator function is implemented as a non-member function, the left-most operand may be an object of the operator's class, an object of a different class, or a built-in type. Now we discuss it in a detailed manner.

We can always write our operators as non-member functions. As a non-member functions, the binary operators like + gets both the operands as arguments. One thing to take care of while writing non-member functions that they cannot access the private members of classes. Actually, this is just to this reason that we make those non-member functions as **friend**s to the classes whose **private** data members are required to be accessed. But the question arises, can we write a non-member operator function without making it a **friend** of a class. The answer to this question is yes; If there are **public** member functions to access the private data members of the class then they serve the purpose. In this case of **Complex** class, let's say we have two public member functions:

```
double real( );
double imaginary( );
```

to access the **private** data members **real** and **imag** respectively. Then we can write non-member **operator +** function as:

```
Complex   operator   + (Complex&  c1, Complex&  c2)
{
        Complex   temp;
        temp.real ( c1.real()   +   c2.real() );
        temp.imaginary ( c1.imaginary()   +   c2.imaginary() );
        return temp;
}
```

But this non-member operation functions without declaring a **friend** of the class is definitely slower than the member function or a **friend** one. The reason for this is obvious from the code that it is making three additional function calls of **real()** and **imaginary()** for each private data member. Also it is not easy to write as compared to member functions. Therefore, it is recommended to write the member functions for operators instead of non-members.

Let's take an example where the operators are performing a non-arithmetical operation. We are writing a class **String** for strings manipulation as:

```
class String
{
   private :

      char   string [ 30 ] ;

   public :

      String ( )
      {
          strcpy ( string , "" ) ;
      }

      void  getString ( )
      {
         cout  <<  "Enter the String : " ;
         cin  >>  string ;
      }

      void displayString ( )
      {
        cout  <<  "The String Is : "  <<  string  <<  endl ;
      }

      // Declaration (prototype) of overloaded sum operator

      String  operator + ( String & s ) ;
};
```

We want to write + operator to concatenate two strings. Firstly, we will see the operator's behavior in ordinary context (behavior with primitive variables for example) and try to implement the same behavior for this class. We want to

vubookshoppk.com

concatenate two strings (two **String** objects) and then assign the resultant string to a new **String** object. Here is how we will write + operator function.

```
String String :: operator + ( String &s )
{
   String temp;                        // Declared object temp of String type
   strcpy ( temp.string , "" );        // Initialized the temp with empty string
   strcat ( temp.string , string );    // Concatenated the driving object's string to //
temp object
strcat ( temp.string , s.string );     // Concatenated the argument's string to the
// temp object
   return   temp;                      // Returned the temp object
}
```

As you might have guessed already, the **String** object on the left will be the one to drive this + operation and the second String object on the left of + will be passed as an argument to this function. Note that we are not doing the error checking here, the size of the resultant string **temp** may increase the array size **30** (the array size defined in the class).

Example Program 1

Rudimentary implementation of a class named **Complex** class to cater complex numbers. A + operator function has been implemented to add two complex numbers.

```
/* This program implements the basic class for complex numbers and demonstrates +
operator function */

#include <iostream.h>

class Complex
{
   private :
      double   real ;     // Real Part
      double   imag ;   // Imaginary Part

   public :
      /* Parameterless Constructor */
      Complex ( )
      {
         cout   <<   "\n Parameterless Constructor called ..." ;
      }

      /* Parameterized Constructor */
      Complex ( double  r, double  i )
      {
         cout   <<   "\n Parameterized Constructor called ...";
         real   =   r ;
         imag   =   i ;
      }
```

vubookshoppk.com

```
    /* Setter of real data member */
    void   real ( double  r)
    {
      real   =   r ;
    }

    /* Getter of the real data member */
    double   real ( )
    {
         return real ;
    }

    /* Setter of the imag data member */
    void   imaginary ( double  i )
    {
      imag = i ;
    }

    /* Getter of the imag data member */
    double   imaginary ( )
    {
      return   imag ;
    }

    /* A Function to display parts of a Complex object */
    void   display ( )
    {
      cout   <<  "\n\n Displaying parts of complex number ...";
      cout   <<   "\n Real Part : " << real << endl ;
      cout   <<   " Imaginary Part : " << imag << endl ;
    }

    /* Declaration (prototype) of overloaded sum operator */
    Complex   operator  + ( Complex & c2 ) ;

};

Complex  Complex  ::  operator + ( Complex  & c1 )
{
  cout   <<   "\n Operator + called ...";
  Complex   temp ;
  temp.real   =   real   +   c1.real ;
  temp.imag   =   imag   +   c1.imag ;
  return   temp ;
}

void main ( )
{
```

vubookshoppk.com

```
    Complex c1 ( 1 , 2 ) ;    // Consturct an object using the parameterized constructor
    Complex c2 ( 2 , 3 ) ;    // Consturct another object using the parameterized
                              // constructor

    Complex result ;  // Construct an object using a parameterless constructor

    result = c1 + c2 ;  // Call the Operator + to add two complex numbers (c1 & c2)
                        // and then assign the result to 'result' object

     result.display ( ) ; // Display the result object contents

}
```

The output of the program is as follows:

```
Parameterized Constructor called ...
Parameterized Constructor called ...
Parameterless Constructor called ...
Operator + called ...
Parameterless Constructor called ...

Displaying parts of complex number ...
Real Part : 3
Imaginary Part : 5
```

The + operator function can be enhanced to return reference of Complex object. We can also implement += operator. += operator and the enhanced operator + are implemented as:

```
Complex &   Complex  ::  operator + ( Complex &  c1 )
{
   real   =   real  +  c1.real ;
   imag   =   imag  +  c1.imag ;
   return   *this;
}

// Declaration (prototype) of overloaded sum assignment operator definition
Complex & Complex :: operator += ( Complex & c2 )
{
   real   +=   c2.real ;
   imag   +=   c2.imag ;
   return   *this;
}
```

Example Program 2

Rudimentary Implementation of String class to manipulate strings. It uses + operator to concatenate strings.

```
/* This program implements the basic class for strings and demonstrates + operator
function to concatenate two strings*/
```

vubookshoppk.com

```
#include   <iostream.h>
#include   <string.h>

class   String
{
   private :

      char    string [ 30 ] ;   // Array to store string

   public :

      /* Parameterless Constructor */
      String ( )
      {
           strcpy ( string , "" ) ;
      }

      /* Getter function of string */
      void   getString ( )
      {
          cout  <<  "Enter the String: " ;
          cin   >>  string ;
      }

      /* Function to display string */
      void   displayString ( )
      {
         cout   <<   "The String is : "   <<   string   <<   endl ;
      }

      // Declaration (prototype) of overloaded sum operator

      String   operator  + ( String & s ) ;
};


String   String  ::  operator + ( String &s )
{
   String   temp ;
   strcpy ( temp.string , "" ) ;
   strcat ( temp.string , string );
   strcat ( temp.string , s.string );
   return   temp;
}

void main ( )
{
     String   string1 , string2 ;          // Declared two String objects
     string1.getString ( ) ;               // Get string for string1 object
     string2.getString ( ) ;               // Get string for string2 object
```

vubookshoppk.com

```
    String hold  =  string1 + string2 ; // Concatenate string1 and string2 and store the
                                        // result in hold object
    hold.displayString ( ) ;            // Display the string
}
```

The output of the above program is as follows:

```
Enter the String: Operator
Enter the String: Overloading
The String is : OperatorOverloading
```

Tips

Operator Overloading is quite similar to Function Overloading.

There are two types of operators to overload: unary and binary.

C++ built-in operators work for built-in (primitve) types but for user defined data types, user has to write his/her own operators.

There are some restriction while performing Operator Overloading. For example, only existing C++ operators are overloaded without creating a new one in the language. Also, it should not impact the type, semantics (behavior), arity (number of operands required), precedence and associativity of the operator.

For binary member operators, operands on the left drives (calls) the operation.

Operator functions written as non-members but friends of the class, get both the operands as their arguments.

Operators can be written as non-members and even without making them friends. But this is tedious and less efficient way, therefore, it is not recommended.

vubookshoppk.com

Lecture No. 32

Reading Material

Deitel & Deitel - C++ How to Program Chapter. 8

8.6, 8.7, 8.12

Summary

18) Recap
19) Overloading Minus Operator
20) Operators with Date Class
21) Unary Operators

vubookshoppk.com

Recap

Before further discussing the concept of the 'Overloading', we will recapture the things dilated upon in the previous lecture. It is necessary to know that new operators i.e. new symbols cannot be introduced. Only existing symbols can be overloaded. Overloading of operators is exactly like writing functions. However, one should remain close to the original meaning of the operator. Similarly, it is good not to define something in opposite terms e.g. 'plus operator is doing subtraction or multiplication operator carrying out division'. We can do that but it will ultimately a bad thing for a programmer. It makes our program practically unreadable and mis-interpretable. Under operator overloading technique, the binary and unary operators will remain unchanged that is we cannot make unary operator work as binary operator or vice versa. In the previous lectures, we also came across some concepts in terms of driving force behind the operator, e.g. in case of binary operator, the driving force is left hand operand. We have also studied when to use member operators and non-member operators. Today we continue discussion on 'use of operators'.

Overloading Minus Operator

Let's define minus operator (-) with special reference to the complex class. The process of defining the minus operator is quite similar to that of the plus operator. Let's first understand the action of minus operator. It is a binary operator, having two arguments. In this case, both the arguments will be complex numbers. When we subtract two complex numbers, it always return a complex number. Here the subtraction of complex numbers is defined as, 'subtract the real part from real part and subtract the imaginary part from the imaginary one". So a member operator will look like as under:

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Complex operator – (Complex c)

As we are defining it as a member operator, only one argument will be passed to it. It is going to be on the right hand side of the minus operator. The left-hand-side will call this as it is already available to this function. In the body, we will declare a temporary Complex number. This means that the real part of this temporary complex number is the difference of the calling Complex number and the Complex number passed as argument i.e.:

tmp.real = real – c.real;

In the next line, we calculate the difference of imaginary part as:

tmp.imag = imag – c.image;

and return the tmp Complex number. By defining, the minus operator does not mean that minus equal operator has also been defined. If we want to overload the minus equal operator (-=), it is necessary to define it. Let's see how the defining process is carried out. Minus equal to operator like the plus equal to operator behaves in the way that the value of calling party (i.e. the complex number which is on the left hand side) will also be changed. So now we will see that the number itself changing the value when it takes part in the minus equal to operator. Again, we will make this a member function. So only one argument will be passed to it. The complex number will be on the right hand side of the minus equal to operator. In the body of the function, there is no need of any temporary complex number as we are going to change the number on the left hand side of the minus equal to operator. We can write it as:

real -= c.real;
imag -= c.image;

Here *c* is the complex number which is passed as an argument. Now the minus equal to (-=) operator, used in the above statements, is an ordinary minus equal to operator for the integers defined by the C++. So this is a classic example of overloading i.e. the operator being overloaded is using the original or basic operator of same type. That is the end of this function. The original number has been changed. We can return its reference. It depends on its usage.

Here is the code:

```
// The minus operator definition
Complex Complex::operator - ( Complex c )
{
   Complex tmp; // defining a temporary var
   tmp.real = real - c.real;
   tmp.imag = imag - c.imag;
   return tmp;
}

// The -= operator definition
Complex Complex::operator -= ( Complex c )
{
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
    real -= c.real ;
    imag -= c.imag ;
}
```

Last time, we discussed the *string* class besides defining the plus operator as joining the two strings. Can we define minus for the *string* class? Is the minus operator relevant to the class *string*? For me it does not. Unless we come with some very artificial definition. Suppose we have a string as "This is a test" and a second string as "test". The subtraction of these two strings means the deletion of a word or words of second string from the first string. It may make some sense in this example. What will happen if the second string contains "My name is xyz". The subtraction of these strings does not make any sense. The thing we need to understand at this point is that every operator does not make sense for every class. Operators should be used only when these make some common sense so that reader can understand it easily. When you add two strings, it makes lot of sense. We can use either cat function or write this plus operator. As subtraction of strings does not make much sense, so it is not advisable to define it. Only define things that are self-explanatory, readable and understandable.

Operators with Date Class

We have so far been using the *Date* class. Let's think what operators make sense for *Date* class. What will be the meaning of plus operator or minus operator? Here we want to remind you a key thing i.e. "Paying attention to detail". Suppose you have some date and want to add some number to it like today's date plus 5. Does that make sense to you? We will get a new date by adding five to today's date i.e. date after five days. Similarly, if we want to subtract, say 10 from today's date, we should get the date of ten days before. Here is the usage of plus and minus which makes some sense. Can we subtract two dates together like subtraction of 1st Jan. 2002 from 15th Oct. 2002. What meaning it will convey? Perhaps nothing.

Let's consider the addition of a number to a date. Adding an integer to some date, according to the definition we will get some date in the future. The *Date* object will be returned from this function. We need a new date after the addition of integer number. We are defining this as a member-function so that the *Date* object that is calling it, will be passed to the function. The integer that is on the right hand side should be passed as an argument. Therefore in the parenthesis, we will have the integer. Now let's discuss it in detail. How can we add an integer to some date? Let's take today's date. Write it in your copy and see how can five be added to it. If you try to add a number to date, there are so many possibilities that can happen. Suppose, today is second day of the current month. After adding five to it, we will get 7th of this month. That was case I. Let's take the case II. Today is 27th of any month. Now what will be the new date after adding five. First thing, which is very obvious, that the month will get changed. But what will be the date of this new month. It depends whether there are 30 days or 31 days in this month. It may be the month of February. Is it the leap year or not? If it is non-leap year, there will be 28 days in February. Otherwise there will be 29 days. What is a leap year? There are rules to determine whether the year is leap year or not. If the year is divisible by four, it will be leap year. Similarly, being a century year, it may be divided by 400. Then again it is a leap year. Now we have seen that there are many cases when we are adding five to 27th of

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

any month. Two things happen. The month is changed and the date changes according to the days in the month. What if it is the 27th of the December? Now you want to add five days. There are 31 days in December, after adding five it will be 1st of next month. We may have declared an array of twelve months. As December is the twelfth month, the last month of the year, so we have to go to first month of the next year. Here the year has also changed. We will also need to increment 1 to year too. It seems very simple that we have to add an integer number of days to some date. It becomes a complex function. Now suppose we have written this complex function and embedded all the rules in it. Then our life will become much easier. Suppose our semester starts from any date. After adding the period of semester, we will get the end date of the semester. We can do date arithmetic. This is a classic example of "paying attention to detail". To use the class for general purposes, we cannot miss even a single case. If you want to publish this class for others to use, you need to pay attention to detail and make sure that your class handles all of the stuff.

Here is the complete code of the program.
File "Date.h"

```
// The Date class is defined here

class Date{
   private:
      int day;
      int month;
      int  year;
      int daysOfMonth(Date d);              // returns the no of days in a month
      static const int daysInMonth[];   // array containing  the 12 month's days
      bool leapYear(int);                        // tells the year is leap year or not

   public:
      Date(int d = 1, int m = 1, int y = 1900); // constructor with default arguments
      void setDate(int, int, int);                                  // set the date with given
arguments
      void display();              // Display the date on the screen
         // operators prototypes
      Date operator ++ ();       // pre increment operator used as ++date1
      Date operator + (int);  // Plus operator used as date1 + 5
};

// The implementation of the date class.
// initializing the no of days, take 0 for month zero.
const int Date::daysInMonth[] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

// Displaying the function on the screen
void Date::display()
{
   cout <<"\nDate:" << day << "-" << month << "-" << year;
}
//constructor of the date
Date::Date(int d, int m, int y)
{
```

vubookshoppk.com

```
    setDate(d, m, y);
}

// setting the date as given arguments
void Date::setDate(int d, int m, int y)
{
    year = y;

    // if month is wrong then set it to 1
    if (month < 1 && month > 12)
        month = 1;
    else
        month = m;
    // if day is wrong then set it to 1
    if (month == 2 && leapYear(y))
        if (d >=1 && d <=29)
            day = d;
        else
            day = 1;
    else
        if( d >= 1 && d <= daysInMonth[month])
                day = d;
        else
                day = 1;
}

// This function return the number of days in a month
int Date::daysOfMonth(Date d)
{
    if (d.month == 2 && leapYear(d.year))  // if leap year then Feb is 29
        return 29;
    else
        return daysInMonth[d.month];
}

// Testing that the year is leap or not.
bool Date::leapYear(int y)
{
    if ( (y%400 == 0) || (y%100 != 0 && y%4 == 0))
        return true;
    else
        return false;
}

// + operator overloaded for the date. Used as date1 + 5
Date Date::operator + (int numberOfDays)
{
    for (int i = 1; i <= numberOfDays; i++)
    {
        ++(*this);    // calling the pre increment operator
```

vubookshoppk.com

```
    }
    return *this;
}

// Pre increment operator
Date Date::operator ++ ()
{
    if (day == daysOfMonth(*this) && month == 12) // end year
            {
             day = 1;
             month = 1;
             ++year;
            }
            else if(day == daysOfMonth(*this))   // end month
            {
             day = 1;
             ++month;
            }
            else      // not the last day of the month
            {
             day++;
            }
}
```

The main program is:

```
#include <iostream.h>
#include "date.h"

void main()
{
  Date  d1 (26, 12, 2002), d2(28,2 ,2000), d3;
  d1.display();
  ++d1;
  cout << "\nAfter adding 1 day, the date is ";
  d1.display();
  cout << endl;
  d2.display();
  d2 = d2 + 5;
  cout << "\nAfter adding 5 days to the above date";
  d2.display();
 }
```

Output of the program:

```
Date:26-12-2002
After adding 1 day, the date is
Date:27-12-2002

Date:28-2-2000
After adding 5 days to the above date
```

vubookshoppk.com

Date:4-3-2000

Similarly we may have a counter-function that subtracts some number from the date. This is the same but of reverse nature. Suppose it is Jan 3rd and we have to subtract ten days from it. The month will be changed to December while the year is going to be decremented by 1. To determine the date of December, we need to know the number of days in December and count backwards. Now we don't need the number of days of current month. Rather the number of days in previous month is important. Suppose it is 3rd of March and subtract seven from it. What will be the date? Now you have to do complex arithmetic and take care of all the cases. It is very complicated but having only one time effort. Date arithmetic is very important and common in business applications. If someone applies for vacations, you just have to enter that this person is going on leave from this date for ten days and you will know his date of re-joining the duty. If someone works on daily wages and paid after a week. Someday, he comes and says that he is going on vacations. We need to calculate the number of days from the day of last payment to to-date. It is simple date arithmetic. Writing a *Date* class with these appropriate operators overloaded will be very useful exercise. It adds to your overall programming vocabulary.

There are two kinds of programming vocabulary. One is the keywords of C/C++ etc while the second is higher-level vocabulary. What sort of vocabulary we have in our tool- box. In the first part of this course, we have learned how to write loops, nested loops etc. We learn to handle matrices and vectors using those rudimentary rules. Now if you think about that we have written a matrix class and a member function *inverseOfMatrix()*. We can use this function again and again. Similarly in the *Date* class, we can put in some rudimentary calculations on date arithmetic. Add or subtract number of days from some date. These are very useful functions. In the daily wages example, you need to subtract a date from a date. Now we need to overload the minus operator again with date minus date. First we overload the minus operator with date minus some integer number. There may be two versions of minus operator. Here, you have to work in detail. Subtracting a date from another date is relatively non-trivial. As a programming idea, you can think that subtracting two dates involves huge calculations. Can we perform some logical tests here? If we want to implement *date1 – date2* while *date1* is smaller than *date2*. The first question is do we want to return a negative number. Let's say we want this, then *date1 – date2* can return a negative number. So it can return a negative number or zero (if the dates are identical) or positive number (the number of days). How we can implement this functionality? One way to do it is with the help of calendar. Under this method, we will start a loop till the other date is got. Then by reading the loop counter, we can tell the difference in days. It is a good idea. But for that, we need a calendar somewhere. If the dates are in different years, we will have to ensure the availability of calendar of next year. Think about it and try to write this function.

Now what about the plus-operator for two dates? Minus operator for strings did not make a lot of sense. Similarly, the plus operator for two dates does not make much sense. We can add some number to date. But how can we add a date to some other date. There is no logical and straight forward answer to this. So we don't define such a function. The meaning of our operator should be obvious. You can write whatever you want in the function. But it is bad idea. The idea of this exercise is to pay attention to detail. Think of all the various things that can happen. Tabulate them,

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

determine the logic and then start programming. Don't start typing your program before your brain has come up to the same point. First analyze the problem, understand it, look at all the cases, draw a flow chart, write pseudo code. Once you are comfortable with this and know what you want to do then start writing your program. The time spending on analyses is arguably the best usage of your time as a programmer. The time you spend on debugging and removing errors from faulty code is huge. Spending time on good design pays off. You should debug for syntax errors like a semi-colon is missing somewhere. You should not face any logical error at debugging stage because logic errors are very hard to track. You may just not worry about the design and start writing code. The program may work for two or three cases. You may declare that you have written the program. When other starts using it on some other case which you did not cater, the program does not work or produces some strange results. There is no syntax error in the program. The compiler compiles it successfully and makes an executable file. Now we have to check the logic. Determining the logic from the code is a million times more difficult than determining code from logic. In this case, analysis will be always followed by design and then code. Please keep this in mind.

Unary Operators

Let's talk about unary operators. Unary operators take one argument like *i*++ or *i*-- (Post Increment or post decrement operators for integers) or ++i, --i (Pre increment or pre decrement operator). You can't make unary operator as binary operator or binary operator as unary. Let's overload unary operator in the *Date* class. We want to overload ++. This operator should add a day in the current date. When we say *++date1 or date1++*, it should get tomorrow's date. This is same as *date1 +=1 or date1 = date1 + 1*. We simply have to change to tomorrow's date. If this is the member function, it will get the date object automatically. The internal structure of the object is available to the function so that it takes no argument. It will return a *Date* object. Its prototype will be as:

```
Date operator ++ (  ); // pre increment operator
```

What will be in the function definition? You have to pay attention to details. The argument that we used in the plus operator, is also applicable here. What will be the next date when we add 1 to the current date. Let's work it out. If it is not the last date of the month, then simply add one to the day. If the date is the last day of the month, then change the month and date to 1st. If the date is the last date of the year, then increment the year too. Suppose we have some function available which returns the days of month given the month number. So if we say *daysOfMonth(6)* it should return 30. The function is intelligent enough that when we say *daysOfMonth(2)* it should return 28 if the year is not leap year. Otherwise, it will be 29. Therefore we have to send it year too along with the month number. We can also pass it the complete *Date* structure as *daysOfMonth(date1);* We will use this function in writing the ++ operator. In a way, the logic is same as we used in the plus operator. Suppose the object *d* is calling this ++ operator as *d*++ where *d* is an object of type *Date*. Therefore the day, month and year data members will be available to this function. In the body of the function, first of all we will check whether this is the last date of the month as:

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
if (day == daysOfMonth ( *this ) )
{
        // this is the last day of the month
        // process accordingly
}
```

In the above condition, we have checked that day is equal to the number of days in the month or not. If the condition returns true it means that this is the last day of the month. Here we have used *this* to pass the object (current object) to the function *daysOfMonth*. '*this'* pointer is implicitly available to every member function and *this pointer* points to the current object. As per requirement of the program, we have written *d++* where *d* is the object of type *Date.* We are not using the object *d* in the program. This object is itself available.

Now the data of object *d* is available in the function as day, month or year. The object *d* is itself present either from its member data (day, month, year) or through the '*this* pointer' which points to the current object. We can also expand the definition of the function *daysOfMonth( )* as *daysOfMonth(int day, int month, int year)*. If the given day is the last day of the month, we will increment the month. Before doing this, we need to check whether this is the last month or not. Therefore we have to introduce another nested 'if' condition. The code segment will now be as:

```
if (day == daysOfMonth ( this ) )
{
        // this is the last day of the month
    if (month < 12)
        {
            day = 1;
            month++;
        }
        else  // this is the last month i.e. December
        {
                day = 1;
                month = 1;
                year++;
        }
}
else        // not the last day of the month
{
        day++;
}
```

The ++ operator simply adds one to the date of the calling object. We define it as member function. Therefore, no argument is needed. We can make it non-member but have to pass it a *Date* object.

To distinguish the pre increment operator with post increment operator, an int argument is passed to it. The prototype of post increment operator for Date is:

```
Date operator ++ (int  ); // post  increment operator
```

vubookshoppk.com

Here we don't need to use this int argument. The implementation is same as pre increment operator as in both cases we want to add 1 to the date.

Can we implement the plus operator using this function? We can write the plus operator in some new fashion. We pass it a positive integer number, which has to be added to the date. We can write a loop in the plus operator. The loop condition will be as *i* < *number* where *number* is the argument passed to it. So in the program if we have written as *date1+5;* the loop will run for five times and in the body of the loop we have ++*date1*. Suddenly our complicated logic has been boiled down to simple as incremented by 1. This is the classic example of code reuse.

We don't know who is going to use this code. Nobody is perfect. But we should think before writing the program about the structure, interface, the setters and getters and the operators to be overloaded. The thumb rule is if the meaning of + and ++ operator is same as in ordinary arithmetic, then + operator can be used in the ++ operator. Keep in mind that we can call a function from another function. This is a good example of code reuse. We can call the + operator as many times as needed. The *daysOfMonth* is a member function and it is used in ++ operator function. '+ operator' is a member function, used in ++ operator. We are building a hierarchy. Suppose there is some small logical error in the code and the *daysOfMonth* is not returning the correct value. This will effect the + operator as well as ++ operator. When we remove that error, then + and ++ operator both will be corrected. Moral of the story is that whenever we write some code, it is better to see whether we are rewriting some code in the same class. If we are calculating the number of months at two places or determining the leap year at two places, then try to combine this in such a way that things should be calculated at one place. That piece of code may become some utility function. This will not be called from outside the class so we will put this function in the private area. But the member functions can call it. We will make the *daysOfMonth()* as a private member function of the class. It will return the days of the month having checked whether this is leap year or not. Using this utility function, we have written + and ++ operator function. Don't repeat code inside a class. Make it a general rule. Make a function for the repeated code and call it where needed. For efficiency and speed, we can repeat the code. For this, we start using macros. It means that if you put all your logic in a single place and then reuse it. You will get lot of safety and security with this. A correction at one place will make the behavior of the whole class correct.

Let's see another interesting function of the *Date* class. Sometimes, we need to compare two dates i.e. whether a date is greater or less than the other date. In other words, the comparison operator is applied. Comparison operators <, >, <=, >=, == can also be overloaded. How do we determine whether *date1* is greater than *date2*? First of all, what will be its return type. Return type has to be either true or false. It says *date1* is greater than *date2* or *date1* is not greater than *date2*. Let's introduce another keyword *bool*. It is a new data type. It is very simple, it only takes two values true or false. So, the return type of greater than operator (>) is *bool*. The prototype of this member function is as:

```
bool operator > (Date d);
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

The argument *d* is the *Date* object that is on the right side of the greater than sign. The left hand side *Date* object is available to this as this is the member operator of the class. Before writing the code, think abut the logic. We have to determine that the calling date is greater than the date *d* or not. If the year of current date is greater than date d, will the current date greater than date *d*? Certainly, it will be so. If the year is greater, obviously date is greater. If the years of both the dates are equal, then we have to check whether the month of the current date is greater than date *d* or not. If the month of the current date is greater than the date *d*, current date is greater. If the months are also equal, we will compare the days. It's a very simple hierarchical logic. To be able to write this logic cleanly, you should write case by case on paper. Analyze it thoroughly. The logic can be written in reverse too. If the year of the date *d* is greater than the current date, return false and so on. So we can go either true, true, true or false, false, false logic. You will find the false logic quicker. We can use if, 'else if' structures. Return type of this function is 'boolean'. Suppose that in our calling function we have two *Date* objects as *d1* and *d2*. We will write as ***if(d1 > d2).*** Why should we write this? As our operator is returning true or false and 'if' also needs true or false, we can write very clean and neat code. This greater than operator is a member operator of *Date* class. In this case, we have the return type as boolean and not returning the *Date* object. Is it the violation of any rule? The answer is no. The return type can be anything. It needs not to be the same as the class. It can be anything. The same applies to difference between two dates. The difference between two dates will be an integer. It is still a member function. It simply tells us the number of days between two days. It could be negative or positive but it is an integer. There is no such rule that the member operators should return the object of the same class.

The code of the greater than operator is as follows:

```
// Definition of the greater than operator

bool Date :: operator > ( Date d )
{
    if ( year > d.year )  // if year is greater date is greater
    {
      return true;
    }
    else if ( year == d.year)     //if years are equal check month
    {
        if ( month > d.month )  // if month is greater date is greater
        {
              return true;
        }
        else if ( month == d.month)      // if months are equal check dates
        {
            if(day > d.day)
              return true;
            else                                          // otherwise return false
              return false;
        }
        else
           return false;
    }
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
    else
    {
      return false;
    }
}
```

Now you can write all the comparison operator of the *Date* class. The comparison operators are greater than, greater than or equal to, equal to, less than, less than or equal to. If you are writing one, you might want to write all of them. Now we have expanded the *Date* class enough. As an exercise, write your own *Date* class. Keep in mind the principles. What should be in the *Date* class? How should we set its values? How many constructors we need? Do we need a destructor or default destructor is enough. After this, define its public interface that is the member functions that are visible from outside. What operators should be overloaded?

In the program, if we say *date1 + 5,* we know that we will get a date which is five days later. What will happen if we write *5 + date1*? The situation like this may happen. You have published your *Date* class and someone wants to use it in this fashion. Here we have an integer, the plus operator and a *Date* object. It should return an object of type *Date*. To make this work properly, we need to have another operator. You will have to look at the set of operators needed for this class. List them out and write down their behavior. Be very clear what you expect them to do and start writing the class. How can we implement *integer + date*? On the left hand side, we have an integer. If the integer is at the left side, it can't be a member function. Member function is always called by object. Here object is not calling the function. Rather integer is calling. So it has to be a friend function that is sitting outside. It will get two arguments, integer and *Date*. As this is the friend function, the internal structure of the *Date* will be available to it. You will create a new *Date* object based on the given *Date* object and the integer and return it. We have seen that member functions are returning integers or Boolean. Here, a non-member function is returning an object of *Date* class. When we have listed out comprehensively that what will be the interface of our class. Which functions and operators will be visible from outside? When we have written the behavior of our class on paper, it is good to start writing the code. You may have to write a lot of code for this class. Once we have compiled the code and have object file, then anyone can use *Date* object. There will be no problem. We will include the "date.h" file in the program and use and manipulate the *Date* objects. We can use its operators, member functions etc very easily. The effort we put in writing this code does not go waste. It will provide a lot of ease in the main program. The biggest advantage is the encapsulation that has happened. All of the logic that was needed to manipulate the object of class *Date* is now encapsulated in that class. In case of any problem in the behavior of the class, we will need to correct the class, compile it. In the conventional function programming or structured programming, this logic has been split at different locations of the program. It was everywhere. Different things have been embedded at different points. In the function oriented programming, we have written a lot of functions at different locations. Here we have a new data type as *Date*. All the date related functions are at one place. We are encapsulating all the functionalities in the *Date* class that is another reason for doing all of the homework, all the thinking before we write the code. No one can determine all the usage of the *Date* class. If you start determining all the usage of *Date* class and writing the definition of the *Date* class for the last six months, this will be impractical. You

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

would want to keep it within limits but do the homework then you write it. Now you can reuse it as many times as you want.

We need a friend operator when the driving thing is not the object of the class like *integer + date*. The operator is derived by integer. Here, we use a friend function. There are instances where friend operators are used to manipulate two different classes. A classic example in the mathematics is of the multiplication of vector and matrix. If we have to multiply a vector with a matrix, the multiplication operator will be friend of both the classes. It will get both vector and matrix as arguments and manipulate them. Keep in mind that friend operators in a way can also be used to glue two classes. The disadvantage of clubbing them together is that they become interlinked. Write your own overloaded operators and see how they work.

vubookshoppk.com

Lecture No. 33

Reading Material

Deitel & Deitel - C++ How to Program Chapter. 7, 8

7.5, 8.1, 8.2, 8.7, 8.9

Summary

22) Operator Overloading
23) Assignment Operator
24) Example
25) this Pointer
26) Self Assignment
27) Returning this Pointer from a Function
28) Conversions
29) Sample Program (conversion by constructor)

vubookshoppk.com

Operator Overloading

As earlier discussed, overloading of operators is carried out in classes on some occasions to enable ourselves to write a code that looks simple and clean. Suppose, there is a class and its two objects, say *a* and *b*, have been defined. Addition of these objects in the class by writing *a* + *b* will mean that we are adding two different objects (objects are instance of a class which is a user defined data type). We want our code to be simple and elegant. In object base programming, more effort is made in class definitions, as classes are data types that know how to manipulate themselves. These know how to add objects of their own type together, how to display themselves and do many other manipulations. While discussing *date* class in the previous lecture, we referred to many examples. In an example, we tried to increment the 'date'. The best way is to encapsulate it in the class itself and not in the main program when we come around to use the *date* class.

Assignment Operator

At first, we ascertain whether there is need of an assignment operator or not? It is needed when we are going to assign one object to the other, that means when we want to have expression like *a* = *b*. C++ provides a default assignment operator. This operator does a member-wise assignment. Let's say, we have in a structure of a class three integers and two floats as data members. Now we take two objects of this class *a* and *b* and write *a* = *b*. Here the first integer of *a* will have the value of first integer of *b*. The second will have the value of second integer and so on. This means that it is a

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

member-wise copy. The default assignment operator does this. But what is to do if we want to do something more, in some special cases?

Now let's define a *String* class. We will define it our self, without taking the built in *String* class of C. We know that a string is nothing but an array of characters. So we define our *String* class, with a data member buffer, it is a pointer to character and is written as **buf* i.e. a pointer to character (array). There are constructors and destructors of the class. There is a member function *length* that returns the length of the string of the calling object. Now we want an assignment operator for this class. Suppose we have a constructor that allows placing a string into the buffer. It can be written in the main program as under:

String s1 ("This is a test") ;

Thus, an object of *String* has been created and initialized. The string "This is a test" has been placed in its buffer. Obviously, the buffer will be large enough to hold this string as defined in our constructor. We allocate the memory for the buffer by using *new* operator. What happens if we have another *String* object, let's say s2, and want to write *s2 = s1* ; Here we know that the buffer is nothing but a pointer to a memory location. If it is an array of characters, the name of the array is nothing but a pointer to the start of the memory location. If default assignment operator is used here, the value of one pointer i.e. *buf* of one object will be assigned to *buf* of the other object. It means there will be the same address in the both objects. Suppose we delete the object *s1*, the destructor of this object will free the allocated memory while giving it back to the free store. Now the *buf* of *s2* holds the address of memory, which actually has gone to free store, (by the destructor of s1). It is no longer allocated, and thus creates a problem. Such problems are faced often while using default assignment operator. To avoid such problems, we have to write our own assignment operator.

Before going on into the string assignment operator, let's have a look on the addition operator, which we have defined for strings. There is a point in it to discuss. When we defined addition operator for strings, we talked about that what we have to do if we want to add (concatenate) a string into the other string. There we had a simple structure i.e. there is a string defined *char buf* with a space of 30 characters. If we have two string objects and the strings are full in the both objects. Then how these will be added? Now suppose for the moment that we are not doing memory allocation. We have a fixed string buffer in the memory. It is important that the addition operator should perform an error check. It should take length of first string , then the length of second string, before adding them up, and check whether it is greater than the length defined in the String (i.e. 30 in this case). If it is greater than that, we should provide it some logical behavior. If it is not greater, then it should add the strings. Thus, it is important that we should do proper error checking.
Now in the assignment operator, the problem here is that the *buf*, that we have defined, should not point to the same memory location in two different objects of type *String*. Each object should have its own space and value in the memory for its string. So when we write a statement like *s2 = s1*, we want to make sure that at assignment time, the addresses should not be assigned. But there should be proper space and the strings should be copied there. Let's see how we can do this.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Now take a look on the code itself. It is quiet straight forward what we want to do is that we are defining an assignment operator (i.e. =) for the String class. Remember that the object on left side will call the = operator. If we write a statement like *s2 = s1*; *s2* will be the calling object and *s1* will be passed to the = operator as an argument. So the data structure of *s2* i.e. *buf* is available without specifying any prefix. We have to access the *buf* of *s1* by writing *s1.buf*.

Suppose *s2* has already a value. It means that if *s2* has a value, its buffer has allocated some space in the memory. Moreover, there is a character string in it. So to make an assignment first empty that memory of *s2* as we want to write something in that memory. We do not know whether the value, we are going to write, is less or greater than the already existing one. So the first statement is:

delete buf ;

Here we write *buf* without any prefix as it is *buf* of the calling object. Now this buffer is free and needs new space. This space should be large enough so that it can hold the string of *s1*. First we find the length of the string of *s1* and then we use the *new* operator and give it a value that is one more than the length of the buffer of *s1*. So we write it as:

buf = new char[length + 1] ;

where *length* is the length of *s1*. Here *buf* is without a prefix so it is the *buf* of object on the left hand side i.e. *s2*. Now, when the *buf* of *s2* has a valid memory address, we copy the *buf* of *s1* into the *buf* of *s2 with the use of string copy function (strcpy).* We write it as:

strcpy (buf, s1.buf) ;.

Now the *buf* of string of *s1* has been copied in the *buf* of *s2 that* are located at different spaces in the memory. The advantage of it is that now if we delete one object *s1* or *s2*, it will not affect the other one. The complete code of this example is given below.

Example

```
/*This program defines the assignment operator. We copy the string of one object
into the string of other object using different spaces for both strings in the memory.
*/

#include <iostream.h>
#include <string.h>
#include <stdlib.h>
// class definition

class String
{
        private :
```

vubookshoppk.com

```
                char *buf ;
        public:
                // constructors
                String();
                String( const char *s )
                {
                        buf = new char [ 30 ];
                        strcpy (buf,s);
                }
                // display the string
                void display ( )
        {
                        cout << buf << endl ;
        }
                // getting the length of the string
                int length ()const
                {
                        return strlen(buf);
                }
                // overloading assignment operator
                void operator = ( const String &other );
};
// ---------- Assignment operator
void String::operator = ( const String &other )
{
        int length ;
        length = other.length();
        delete buf;
        buf = new char [length + 1];
        strcpy( buf, other.buf );
}

//the main program that uses the new String class with its assignment operator:
main()
{
        String myString( "here's my string" );
        cout << “My string is  =  ” ;
        myString.display();
        cout << '\n';

        String yourString( "here's your string" );
        cout << “Your string is  =  ” ;
        yourString.display();
        cout << '\n';

        yourString = myString;
        cout << “After assignment, your string is  =  ” ;
        yourString.display();
        cout << '\n';
        system ("pause");
```

vubookshoppk.com

```
}
```

Following is the output of the program.

```
My string is = here's my string

Your string is = here's your string

After assignment, your string is =  here's my string
```

The above example is regarding the strings. Yet in general, this example pertains to all classes in which we do memory manipulations. Whenever we use objects that allocate memory, it is important that an assignment operator (=) should be defined for it. Otherwise, the default operator will copy the values of addresses and pointers. The actual values and memory allocation will not be done by it.

Let's go on and look what happens when we actually do this assignment? In the assignment of integers, say we have three integers i, j and k with some values. It is quiet legal to write as *i = j* ; By this ,we assign the value of *j* to *i*. After this we write *k = i* ; this assigns the value of *i* to *k*. In C, we can write the above two assignment statements in one line as follows

k = i = j ;

This line means first the value of *j* is assigned to *i* and that value of *i* is assigned to *k*. The mechanism that makes this work is that in C or C++ every expression itself has a value. This value allows these chained assignment statements to work. For example, when we write *k = i = j* ; then at first *i = j* is executed. The value at the left hand side of this assignment statement is the value of the expression that is returned to the part '*k =*' .This value is later assigned to *k*. Now take another example. Suppose we have the following statement:

k = i = ++j ;

In this statement, we use the pre-increment operator, as the increment operator (++) is written before *j*. This pre-increment operator will increment the value of *j* by 1 and this new value will be assigned to *i*. It is pertinent to note that this way *++j* returns a value that is used in the statement. After this, the value of *i* is assigned to *k*. For integers, it is ok. Now, how can we make this mechanism work with our *String* objects. We have three *String* objects *s1*, *s2* and *s3*. Let's say there is a value (a string) in the buffer of *s1*. How can we write *s3 = s2 = s1*;. We have written *s2 = s1* in the previous example in assignment operator. We notice that there is no return statement in the code of the assignment operator. It means that operator actually returns nothing as it has a void return type. If this function is not returning any thing then *s3 = s2 = s1* ; cannot work. We make this work with the help of *this* pointer.

this Pointer

Whenever an object calls a member function, the function implicitly gets a pointer from the calling object. That pointer is known as *this* pointer. '*this*' is a key word. We cannot use it as a variable name. '*this*' pointer is present in the function, referring to

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

the calling object. For example, if we have to refer a member, let's say *buf*, of our *String* class, we can write it simply as:

b*uf ;*

That means the *buf* of calling object is being considered. We can also write it as

this->buf ;

i.e. the data member of the object pointed by *this* pointer is being called. These (*buf* and *this->buf*) are exactly the same. We can also write it in a third way as:

*(*this).buf* ;.

So these three statements are exactly equivalent. Normally we do not use the statements written with *this* key word. We write simply *buf* to refer to the calling object.

In the statement *(*this).buf* ; The parentheses are necessary as we know that in *object.buf* the binding of dot operator is stronger than the *. Without parentheses the *object.buf* is resolved first and is dereferenced. So to dereference *this* pointer to get the object, we enforced it by putting it in parentheses.

Self Assignment

Suppose, we have an integer '*i*. In the program, somewhere, we write *i = i* ; It's a do nothing line which does nothing. It is not an error too. Now think about the String object, we have a string *s* that has initialized to a string, say, 'This is a test'. And then we write *s = s* ; The behavior of equal operator that we have defined for the String object is that it, at first deletes the buffer of the calling object. While writing *s = s* ; the assignment operator frees the buffer of *s*. Later, it tries to take the buffer of the object on right hand side, which already has been deleted and trying to allocate space and assign it to *s*. This is known as self-assignment. Normally, self -assignment is not directly used in the programs. But sometimes, it is needed. Suppose we have the address of an object in a pointer. We write:

*String s, *sptr* ;

Now *sptr* is a pointer to a *String* while *s* is a *String* object. In the code of a program we can write

sptr = &s ;

This statement assigns the address of *s* to the pointer *sptr*. Now some where in the program we write *s = *sptr* ; that means we assign to *s* the object being pointed by *sptr*. As *sptr* has the address of *s* , earlier assigned to it. This has the same effect as *s = s* ;. The buffer of *s* will be deleted and the assignment will not be done. Thus, the program will become unpredictable. So self-assignment is very dangerous especially at a time when we have memory manipulation in a class. The *String* class is a classic example of it in which we do memory allocation. To avoid this, in the equal operator (operator=), we should first check whether the calling object (L.H.S.) and the object

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

being gotten (R.H.S.) are the same or not. So we can write the equal operator (operator=) as follows

```
void String::operator=( const String &other )
{
        if( this == &other )
                return;
delete buf;
length = other.length;
buf = new char[length + 1];
strcpy( buf, other.buf );
}
```

Here above, the statement *if (this == &other)* checks that if the calling object (which is referred by *this*) is the same as the object being called then do nothing and return as in this case it is a self assignment. By doing this little change in our assignment operator, it has become safe to use. So it is the first usage of *this* pointer that is a check against self assignment.

Returning *this* Pointer From a Function

Now lets look at the second use of *this* pointer. We want to do the assignment as

```
s3 = s2 = s1 ;
```

In this statement the value of *s2 = s1* is assigned to *s3*. So to do this it is necessary that the assignment operator should return a value. Thus, our assignment operator will expand so that it could return a value. The assignment operator, till by now copies the string on right hand side to the left hand side. This means *s2 = s1* ; can be done by this. Now we want that this *s2* should be assigned to *s3*, which can be done only if *s2 = s1* returns *s2* (an object). So we need to return a *String* object. Here becomes the use of *this* pointer, we will write as

```
return *this ;
```

Here, in the assignment operator code *this* is referring to the calling object (i.e. s2 in this case). So when we write *return *this* ; it means return the calling object (object on L.H.S.) as a value. Thus *s3* gets the value of *s2* by executing *s3 = s2* where *s2* is the value returned by the assignment operator by *s2 = s1*; Thus the complete assignment operator (i.e. operator= function) that returns a reference to an object will be written as under

```
String &String::operator=( const String &other )
{
        if( &other == this )    //if calling and passed objects are
                return *this;           // same then do nothing
and return
        delete buf;
        length = other.length;
    buf = new char[length + 1];
    strcpy( buf, other.buf );
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
        return *this;
    }
```

Now, here the first line shows that this *operator=* function returns a reference to an object of type *String* and this function takes a reference as an argument. The above version of *operator=* function takes a reference as an argument. First of all it checks it with the calling object to avoid self assignment. Then it deletes the buffer of calling object and creates a new buffer large enough to hold the argument object. And then at the last it returns the reference of the calling object by using *this* pointer.
With this version of the assignment operator (operator= function) we can chain together assignments of *String* objects like *s3 = s2 = s1* ;
Actually, we have been using *this* pointer in chained statements of *cout*. For example, we write

cout << a << b << c ;

Where *a*, *b*, and *c* are any data type. It works in the way that the stream of *cout* i.e. << is left associative. It means first *cout << a* is executed and to further execute it, the second << should have *cout* on its left hand side. So here, in a way, in this operator overloading, a reference to the calling object that is *cout* is being returned to the stream insertion <<. Thus a reference of *cout* is returned, and (as a reference to *cout* is returned) the << sees a *cout* on left hand side and thus the next *<< b* is executed and it returns a reference to *cout* and with this reference the next *<< c* works. This all work is carried out by *this* pointer.

We have seen that value can be returned from a function with *this* pointer. We used it with assignment operator. Lets consider our previous example of *Date* class. In that class we defined increment operator, plus operator, plus equal operator, minus operator and minus equal operator. Suppose we have *Date* objects *d1*, *d2* and *d3*. When we write like *d2 = d1++* ; or *d2 = d1 + 1* ; here we realize that the + operator and the ++ operator should return a value. Similarly, other operators should also return a value. Now let's consider the code of *Date* class. Now we have rewritten these operators. The difference in code of these is not more than that now it returns a reference to an object of type *Date*. There are two changes in the previous code. First is in the declaration line where we now use & sign for the reference and we write it like

Date& Date::operator+=(int days)

We write this for the all operators (i.e. +, ++, - and -=). Then in the function definition we return the reference of the left hand side *Date* object (i.e. calling object) by writing

*return *this* ;*

Now we can rewrite the *operator+=* of the *Date* class as follows.
The declaration line in the class definition will be as

Date& operator+=(int days);

And the function definition will be rewritten as the following.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
Date& Date::operator+=(int days)   // return type  reference to
object
{
        for (int i=0; i < days; i++)
        *this++;
        return *this;    // return reference to object
}
```

This concludes that whenever we are writing arithmetic operator and want that it can be used in chained statements (compound statements) then we have to return a value. The easiest and most convenient way of returning that value is by returning a reference to the calling object. So, by now we can easily write the statements of *date* object, just like we write for integers or floats. We can write

```
        date2 = date1 + 1 ;
Or      date2 = date1++ ;
```

and so on.

Conversions

Being in the C language, suppose we have an integer *i* and a float *x*. Now in the program we write *x = i* ; As we know that the operations of *int* and *float* are different in the memory. Therefore, we need to do some kind of conversion. The language automatically converts *i* (int) to a float (or to whatever type is on the L.H.S.) and then does the assignment.

Both C and C++ have a set of rules for converting one type to another. These rules are used in the following situations

- When assigning a value. For example, if you assign an integer to a variable of type long, the compiler converts the integer to a long.
- When performing an arithmetic operation. For example, if you add an integer and a floating-point value, the compiler converts the integer to a float before it performs the addition.
- When passing an argument to a function; for example, if you pass an integer to a function that expects a long.
- When returning a value from a function; for example, if you return a float from a function that has double as its return type.

In all of these situations, the compiler performs the conversion implicitly. We can make the conversion explicit by using a cast expression.

Now the question arises that can we do conversion with objects of our own classes. The answer is yes. If we go to the basic definition of a class it is nothing but a user defined data type. As it is a user defined data type, we can also define conversion on it. When we define a class in C++, we can specify the conversions that the compiler can apply when we use instances of that class. We can define conversions between classes, or between a class and a built-in type

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

There is an example of it. Suppose we have stored date in a serial number form. So now it is not in the form of day, month and year but it is now a long integer. For example if we start from January 1, 1900 then 111900 will be the day one, second January will be the day 2, third January will be the day 3 and so on. Going on this way we reached in year 2000. There will be a serial number in year 2000. This will be a single long integer that represents the number of days since a particular date. Now if we have a *Date* object called *d* and an integer *i*. We can write *d = i* ; which means we want that *i* should go in the serial part of *d*. This means convert the integer *i* into *date* object and then the value of *i* should go into the serial number and then that object should be assigned to *d*. We can make this conversion of *Date* object. We do this in the constructor of the class. We pass the integer to the constructor, here convert it into a date, and the constructor then returns a *Date* object. On the other hand, if there is no constructor then we can write conversion function. We have used the conversion operator with *cast*. The way of casting is that we write the name of the cast (type to which we want to convert) before the name of variable. Thus if we have to write *x = i* ; where *x* is a float and *i* is an integer then we write it with conversion function as *x = (float) i* ; The (float) will be the conversion operator. Normally this conversion is done by default but sometimes we have to force it. Now we want to write a conversion operator, which converts an integer to a *Date* object. (here Date is not our old class, it's a new one). We can write a conversion function. This function will be a member of the *Date* class. This function will return nothing, as it is a conversion function. The syntax of this function will be *Date ()* that means now this is a conversion function.

In the body of this function we can write code of our own. Thus we can define the operator, and it will work like that we write within the parentheses the name of the conversion operator. The conversion functions are quiet interesting. They allow us to manipulate objects of different classes. For example, we have two classes one is a truck and other is a car. We want to convert the car to a truck. Here we can write a conversion function, which says take a car convert it into a truck and then assign it to an object of class truck.

Sample Program (conversion by constructor)

Lets take a look at an example in which the conversion functions are being used. There is a class *Fraction*. Lets talk why we called it *Fraction* class. Suppose we have an assignment

double x = 1/3 ;

When we store 1/3 in the computer memory, it will be stored as a double precision number. There is a double precision division of 1 by 3 and the answer 0.33333… is stored. Here the number of 3s depends upon the space in the memory for a double precision number. That value is assigned to a *double* variable. What happens if we multiply that *double* variable by 3, that means we write *3 * x*; where x was equal to 1/3 (0.33333…). The answer will be 0.99999…, whatever the number of digits was. The problem here is that the answer of 3 * 1 / 3 is 1 and not 0.99999. This problem occurs, when we want to represent the numbers exactly. The fraction class is the class in which we provide numerator and denominator to the object and it stores them separately. It will always keep them as an integer numerator and an integer denominator and we can do all kinds of arithmetic with it. Now if 1 and 3 are stored

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

separately as numerator and denominator, how we can add them to some other fraction. We have enough tools at our disposal. One of which is that we can overload the addition operator. We can add 1/3 and 2/5, the result of which is another fraction i.e. numerator and denominator. In this way we have no worries of round off errors, the truncation and conversions of int and floats.

Considering the *Fraction* class we can think of a constructor which takes an integer and converts it into a fraction. We do not want that as a fraction there should be some value divided by zero. So we define the default constructor that takes two integers, one for numerator and one for denominator. We provide a default value for the denominator that is 1. It means that now we can construct a fraction by passing it a single integer, in which case it will be represented as a fraction with the passed integer as a numerator and the default vale i.e. 1 as the denominator. If we have a fraction object *f* and we write *f = 3*; Then automatically this constructor (i.e. we defined) will be called which takes a single integer as an argument. An object of type fraction will be created and the assignment will be carried out. In a way, this is nothing more than a conversion operation. That is a conversion of an integer into a fraction. Thus a constructor that takes only one parameter is considered a conversion function; it specifies a conversion from the type of the parameter to the type of the class.

So we can write a conversion function or we can use a constructor of single argument for conversion operation. We cannot use both, we have to write one or the other. Be careful about this. We don't use conversion functions often but sometimes it is useful to write them. The conversion operators are useful for defining an implicit conversion from the class to a class whose source code we don't have access to. For example, if we want a conversion from our class to a class that resides within a library, we cannot define a single-argument constructor for that class. Instead, we must use a conversion operator.

It makes our code easier and cleaner to maintain. It is important that pay more attention while defining a class. A well-defined class will make their use easy in the programming.
Following is the code of the example stated above.

```
/* This program defines a class Fraction which stores numerator and
denominator of a fractional number separately. It also overloads the
addition operator for adding the fractional numbers so that exact results
can be obtained.
*/

#include <stdlib.h>
#include <math.h>
#include <iostream.h>

// class definition
class Fraction
{
        public:
                        Fraction();
                        Fraction( long num, long den );
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
                        void display() const;
                        Fraction operator+( const Fraction &second ) const;
        private:
                        static long gcf( long first, long second );
                        long numerator, denominator;
};

// ----------- Default constructor
Fraction::Fraction()
{
        numerator = 0;
        denominator = 1;
}
// ----------- Constructor
Fraction::Fraction( long num, long den )
{
        int factor;
        if( den == 0 )
          den = 1;
        numerator = num;
        denominator = den;
        if( den < 0 )
        {
                        numerator = -numerator;
                        denominator = -denominator;
         }
        factor = gcf( num, den );
        if( factor > 1 )
        {
                        numerator /= factor;
                        denominator /= factor;
        }
}
// ----------- Function to print a Fraction
void Fraction::display() const
{
        cout << numerator << '/' << denominator;
}

// ----------- Overloaded + operator
Fraction Fraction::operator+( const Fraction &second ) const
{
        long factor, mult1, mult2;
        factor = gcf( denominator, second.denominator );
        mult1 = denominator / factor;
        mult2 = second.denominator / factor;
        return Fraction( numerator * mult2 + second.numerator * mult1,
                                denominator * mult2 );
}
```

vubookshoppk.com

```
// ---------- Greatest common factor
// computed using iterative version of Euclid's algorithm
long Fraction::gcf( long first, long second )
{
        int temp;
        first = labs( first );
        second = labs( second );
        while( second > 0 )
        {
                temp = first % second;
                first = second;
                second = temp;
        }
        return first;
}

//main program
void main()
{
        Fraction a, b( 23, 11 ), c( 2, 3 );
        a = b + c;
        a.display();
        cout << '\n';
        system("pause");
}
```

The output of the program is as follows

```
91/33
```

Here is an example from the real world. What happens if we are dealing with currency? The banks deal with currency by using computer programs. These programs maintain the accounts by keeping the track of transactions and manipulating deposits and with drawls of money. Suppose the bank declares that this year the profit ratio is 3.76 %. Now if the program calculates the profit as 3.67 % of the balance, will it be exactly in rupees and paisas or in dollars and cents? Normally, all the currencies have two decimal digits after decimal point. Whenever we apply some kind of rates in percentage the result may become in three or four decimal places. The banks cannot afford that the result of 90 paisas added to 9 rupees and 10 paisas become 10 rupees and 1 paisa. They have to accurate arithmetic. So they do not rely on programs that use something like *double* precisions to represent currencies. They would have rather written in the program to treat it as a string. Thus 9 is a string, 10 is a string and the program should define string addition such that the result of addition of the strings .10 and .90 should be 1.00. Thus, there are many things that happen in real world that force us as the programmer to program the things differently. So we do not use native data types.

The COBOL, COmmon Business Oriented Language has a facility that we can represent the decimal numbers exactly. Internally it (the language) keeps them as strings in the memory. There is no artificial computer representation of numbers. Now a day, the languages provide us the facility by which we can define a data type

vubookshoppk.com

according to a specific requirement, as we defined *fraction* to store numerator and denominator separately. By using this *fraction* data type, we never loose precision in arithmetic. The same thing applies to the object like currency, where we can store the whole number part and the fractional part both as separate integers and never loose accuracy. But whenever we get into these classes, it is our responsibility to start writing all of the operators that are required to make a complete class.

Lecture No. 34

Reading Material

Deitel & Deitel - C++ How to Program ***Chapter 8***

vubookshoppk.com

Summary

- Arrays of Objects
- Dynamic Arrays of Objects
- Overloading new and delete Operators
- Example of Overloading new and delete as Non-members
- Example of Overloading new and delete as Members
- Overloading [] Operator to Create Arrays

Arrays of Objects

A class is a user-defined data type. Objects are instances of classes the way *int* variables are instances of *int*s. Previously, we have worked with arrays of *int*s. Now, we are going to work with arrays of objects.

The declaration of arrays of user-defined data types is identical to the array of *primitive* data types.

Following is a snapshot of our veteran *Date* class:

```
/* Snapshot of Date class discussed in previous lectures */
class Date
{
    private:
      int   day, month, year;

    public:
     /* Parameterless constructor, it is created by the compiler automatically when we
```

```
        don't write it for any of our class. */
    Date( )
    {
        cout << "\n Parameterless constructor called ...";
        month = day = year = 0;
    }
    /* Parameterized constructor; has three ints as parameters. */
    Date(int month, int day, int year)
    {
        cout << "\n Constructor with three int parameters called ...";
        this->month = month;
        this->day = day;
        this->year = year;
    }

    ~Date ( )
    {
        cout << "\n Destructor called ...";
    }
    . . .
    . . .
};
```

Consider the example of declaring an array of *10* date objects of *Date* class. In this case, the declaration of arrays will be as under:
Following is the declaration:

Date myDates [10] ;

With this line (when this line is executed), we are creating 10 new objects of *Date* class. We know that a constructor is called whenever an object is created. For every object like *myDate[0], myDate[1],…. myDate[9]*, the constructor of the *Date* class is called. Theimportant thing to know here is that which constructor of Date class is being called to construct objects of the array *myDates*. As we are not doing any initialization of the array objects explicitly, the default constructor (parameterless constructor) of the Date class is called. Remember, the default constructor is defined by the C++ compiler automatically for every class that has no parameterless constructor defined already. In our case of *Date* class, we have defined a parameterless constructor, therefore, the compiler will not generate default constructor automatically.

We can also initialize the array elements at the declaration time. This initialization is similar to that done for native data types. For *int* array, we used to do initialization in the following manner:

int array [10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 } ;

Similarly, we initialize *Date* array while declaring it:

Date yourDate [3] = { Date(10, 24, 1980), Date(06, 14, 1985), Date(07, 09,1986) };

vubookshoppk.com

The above statement will call the parameterized constructor *Date (int month, int day, int year)* of *Date* class to create three objects of *myDate* array. This parameterized constructor carries out initialization of the objects data member (*month, day, year*) with the values supplied as arguments to it.

It will be interesting to know, how the following statement works:

Date myDate [10] = { Date(09, 03, 1970), Date(08, 23, 1974) } ;

We are trying to declare an array of 10 Date objects while supplying only initialization values for the first two elements. At first, we might be doubtful if the statement is compiled successfully. Not only it compiles successfully but also does the initialization of the first two objects (*myDate[0], myDate[1]*). What will happen to the remaining objects in the array? Actually, all the *10* objects are created successfully by the above statement. The parameterized constructor is called for the first two objects (*myDate[0], myDate[1]*) and parameterless constructor is called for the remaining objects (*myDate[2], myDate[3], …, myDate[9]*).

You might have noticed that at the array initialization stage, we have explicitly called parameterized constructor of *Date* for every object. We may specify only the argument when a constructor with only one parameter is called.

```
/* A snapshot of String class discussed in previous lectures */

class String
{
private :
        char   *buf ;
public:
        // Constructors
        String ();
        String( const char   *s )
        {
              buf  =  new  char [ 30 ];
              strcpy (buf,s);
        }
        . . .
        . . .
};
```

For example, in the above-mentioned case of *String* class, we have a constructor that is accepting one argument of type *char* *. While writing our code, we can declare and initialize an array of *String*s as follows:

```
String message [10] = {      "First line of message\n",
                             "Second line of message\n",
                             String( "Third line of message\n"),
                             String( )
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
};
```

See the initializing arguments for first two objects i.e, (*message[0]*, *message[1]*) in the array. Here only one string is being passed. Therefore, for the first two objects, constructor with one parameter of type *char* * of *String* class is called automatically. That constructor is *String (char * str)*. For the third object (*message[2]*), the same constructor with one *char* * as parameter is being called explicitly. For fourth object (*message[3]*), parameterless constructor i.e., *String ( )* is being called explicitly, though, this was optional as parameterless constructor is called up automatically when no initialization is made. As there is no explicit initialization for the remaining six objects, the parameterless constructor is called up automatically.
Can we create arrays of objects dynamically? As usual, the answer is yes. Let's discuss it in detail.

Dynamic Arrays of Objects

Consider the following statement:

```
1.  String  *text ;
2.  text  =  new  String [5] ;
```

In line 1, we have declared a pointer *text* of *String* type.
In line 2, we are creating an array of *5* objects of *String* type. This statement allocates space for each object of the array, calls the parameterless constructor for each object and starting address of the first object is assigned to the pointer *text*.
The important point to be noted here is that in line 2, we can't initialize objects because there is no way to provide initializers for the elements of an array allocated with *new*.
The default constructor (parameterless constructor) is called for each element in the array allocated with *new*. Remember, the default constructor for a class is generated by C++ compiler automatically if it is not defined already in the class definition.
To deallocate these arrays of objects, the *delete* operator is used in the same way as it is used for the native data types.
There are few cautions that should be taken care of while performing these operations of allocation and deallocation with arrays of objects.
Firstly, while deallocating an array allocated with *new* operator, it is important to tell the compiler that an array of objects is being deleted. The brackets (*[]*) are written in our *delete* statement after the *delete* keyword to inform the *delete* operator that it is going to delete an array. The consequences of using the wrong syntax are serious. For example, if we want to delete previously created array of five *String* objects using the following statement:

```
delete  text;   // Incorrect syntax of deleting an array
```

The delete operator in this case will not be aware of deleting (deallocating) an array of objects. This statement will call the destructor only for the object pointed by the ***text*** pointer i.e. *String[0]* and deallocate the space allocated to this object. The requirement is to call the destructor for all the objects inside the array and deallocate the space allocated to all of these objects. But on account of the wrong syntax, only the first object is deleted and the remaining four objects (*String[1], String[2], String[3], String[4]* pointed by *text[1], text[2], text[3], text[4]* respectively) remain

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

in the memory intact. The memory space occupied by these four objects results in *memory leak* as the same program or any other program on the same computer cannot use it unless it is deallocated.
Calling the destructor while destroying an object becomes essential when we have allocated some memory in free store from inside the object (usually from within the constructor).
To destroy an array of objects allocated on free store using the *new* operator, an array equivalent of *delete* operator is used. The array equivalent of *delete* operator is to write empty square brackets after the *delete* keyword (*delete []*). So the correct statement is:

```
delete [] text ;
```

This statement destroys the whole array properly. It calls destructor for each object inside the array and deallocates the space allotted to each object. Actually, by looking at the brackets (*[]*) after *delete*, the compiler generates code to determine the size of the array at runtime and deallocate the whole array properly. Here, it will generate code to deallocate an array of 5 objects of *String* type.

If we create an array of *Date* objects and want to delete them without specifying array operator: It will look as under:

```
// Bad Technique: deleting an array of objects without []
//  for a class that is not doing dynamic memory allocation internally
Date  * ppointments;
appointments  =  new Date[10];
. . .
delete  appointments;  // Same as delete [] appointments;
```

Although, this is good to deallocate an array of objects without specifying array operator ([]) as there is no dynamic memory allocation occurring from inside the *Date* class. But this is a bad practice. In future, the implementation of this class may change. It may contain some dynamic memory allocation code. So it is always safer to use array operator (*[]*) to delete arrays.

Can we overload *new* and *delete* operators? Yes, it is possible to overload *new* and *delete* operators to customize memory management. These operators can be overloaded in global (non-member) scope and in class scope as member operators.

Overloading of new and delete Operators

Firstly, we should know what happens when we use *new* operator to create objects. The memory space is allocated for the object and then its constructor is called. Similarly, when we use *delete* operator with our objects, the destructor is called for the object before deallocating the storage to the object.
When we overload *new* or *delete* operators, it can only lead to a change in the allocation and deallocation part. The call to the constructor after allocating memory while using *new* operator and call to the destructor before deallocating memory while

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

using *delete* operator will be there. These calls to constructors and destructors are controlled by the language itself and these cannot be altered by the programmer.

One of the reasons of overloading *new* and *delete* operators can be their limited current functionality. For example, we allocate space on free store using the *new* operator for *1000 int*s. It will fail and return *0* if there is no contiguous space for *1000 int*s in free store. Rather the free store has become fragmented. The total available memory is much more than *1000 int*s , but in fragments. There is no contiguous segment of at least *1000 int*s. The built-in *new* operator will fail to get the required space but we can overload our own *new* operator to de-fragment the memory to get at least *1000 int*s space. Similarly, we can overload *delete* operator to deallocate memory.

In the embedded and real-time systems, a program may have to run for a very long time with restricted resources. Such a system may also require that memory allocation always takes the same amount of time. There is no allowance for heap exhaustion or fragmentation. A custom memory allocator is the solution. Otherwise, programmers will avoid using *new* and *delete* altogether in such cases and miss out on a valuable C++ asset.

There are also downsides of this overloading. If we overload *new* or *delete* operator at global level, the corresponding built-in *new* or *delete* operator will not be visible to whole of the program. Instead our globally written overloaded operator takes over its place all over. Every call to *new* operator will use our provided *new* operator's implementation. Even in the implementation of *new* operator, we cannot use the built-in *new* operator.
Nonetheless,when we overload *new* operator at a class level then this implementation of *new* operator will be visible to only objects of this class. For all other types (excluding this class) will still use the built-in *new* operator. For example, if we overload *new* operator for our class *Date* then whenever we use *new* with *Date*, our overloaded implementation is called.

```
Date* datePtr = new Date;
```

This statement will cause to call our overloaded *new* operator. However, when we use *new* with any other type anywhere in our program as under:

```
int* intPtr = new int [10];
```

The built-in *new* operator is called. Therefore, it is safer to overload *new* and *delete* operators for specific types instead of overloading it globally.

An important point to consider while overloading *new* operator is the return value when the *new* operator fails to fulfill the request. Whether the operator function will return *0* or throw an exception.

Following are the prototypes of the *new* and *delete* operators:

```
void * operator new ( size_t size ) ;
void operator delete ( void * ptr) ;
```

The *new* operator returns a *void* * besides accepting a parameter of whole numbers *size_t*. This prototype will remain as it is while overloading *new* operator. In the implementation of overloaded *new* operator, we may use *calloc( )* or *malloc( )* for memory allocation and write some memory block's initialization code.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

The delete operator returns nothing (*void*) and accepts a pointer of *void* * to the memory block. So the same pointer that is returned by the *new* operator, is passed as an argument to the *delete* operator. Remember, these rules apply to both if operators (*new* and *delete*) are overloaded as member or non-member operators (as global operators). Importantly, whenever we use these operators with classes, we must know their sequence of events that is always there with these operators. For *new* operator, memory block is allocated first before calling the constructor. For *delete* operator, destructor for the object is called first and then the memory block is deallocated. Importantly, our overloaded operators of *new* and *delete* only takes the part of allocation and deallocation respectively and calls to constructors and destructors remain intact in the same sequence.

Because of this sequence of events, the behavior of these *new* and *delete* operators is different from the built-in operators of *new* and *delete*. The overloaded *new* operator returns *void* * when it is overloaded as non-member (global). However, it returns an object pointer like the built-in *new* operator, when overloaded as a member function.

It is important to understand that these operator functions behave like *static* functions when overloaded as member functions despite not being declared with *static* keyword. *static* functions can access only the *static* data members that are available to the class even before an object is created. As we already know that *new* operator is called to construct objects, it has to be available before the object is constructed. Similarly, the *delete* operator is called when the object has already been destructed by calling destructor of the object.

vubookshoppk.com

Example of Overloading new and delete as Non-members

Suppose we want *new* to initialize the contents of a memory block to zero before returning it. We can achieve this by writing the operator functions as follows:

```
/* The following program explains the customized new and delete operators */

#include   <iostream.h>
#include   <stdlib.h>
#include   <stddef.h>

// ------------- Overloaded new operator
void *   operator  new ( size_t   size )
{
  void *   rtn   =   calloc( 1,  size );  // Calling calloc() to allocate and initialize memory
  return   rtn;
}


// ----------- Overloaded delete operator
void   operator  delete  ( void *   ptr )
{
  free( ptr );   // Calling  free() to deallocate memory
}
void   main()
```

```
{
   // Allocate a zero-filled array
   int  *ip  =  new  int[10];
   // Display the array
   for ( int i  =  0; i  <  10; i ++ )
      cout  <<  " "  <<  ip[i];
   // Release the memory
   delete  [] ip;
}
```

The output of the program is as follows.

```
0 0 0 0 0 0 0 0 0 0
```

Note that the *new* operator takes a parameter of type *size_t*. This parameter holds the size of the object being allocated, and the compiler automatically sets its value whenever we use *new*. Also note that the *new* operator returns a *void* pointer. Any *new* operator we write must have this parameter and return type.
In this particular example, *new* calls the standard C function *calloc* to allocate memory and initialize it to zero.
The *delete* operator takes a *void* pointer as a parameter. This parameter points to the block to be deallocated. Also note that the *delete* operator has a *void* return type. Any *delete* operator we write, must have this parameter and return type.
In this example, *delete* simply calls the standard C function *free* to deallocate the memory.

Example of Overloading new and delete as Members

```
// Class-specific new and delete operators

#include  <iostream.h>
#include  <string.h>
#include  <stddef.h>
const  int  MAXNAMES  =  100;

class  Name
{
public:
  Name( const char *s ) {  strncpy( name,  s,  25 ); }
  void  display()  const {  cout  <<  '\n'  <<  name; }
  void *  operator  new ( size_t  size );
  void  operator  delete( void *  ptr );
  ~Name() {};   // do-nothing destructor
private:
  char  name[25];
};
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
// -------- Simple memory pool to handle fixed number of Names
char   pool[MAXNAMES] [sizeof( Name )];
int   inuse[MAXNAMES];

// -------- Overloaded new operator for the Name class
void *   Name :: operator   new( size_t size )
{
   for( int   p = 0; p < MAXNAMES; p++ )
         if( !inuse[p] )
         {
            inuse[p]   =   1;
            return   pool + p;
         }
   return   0;
}

// --------- Overloaded delete operator for the Names class
void   Name :: operator   delete( void *ptr )
{
   inuse[((char *)ptr - pool[0]) / sizeof( Name )] = 0;
}

void main()
{
   Name *   directory[MAXNAMES];
   char   name[25];
   for( int i = 0; i < MAXNAMES; i++ )
   {
      cout   <<   "Enter name # "   <<   i+1   <<   ": ";
      cin   >>   name;
      directory[i]   =   new   Name( name );
   }
   for( i = 0; i < MAXNAMES; i++ )
   {
      directory[i]->display();
      delete directory[i];
   }
}
```

The output of the above program is given below.

```
Enter name # 1: ahmed
Enter name # 2: ali
Enter name # 3: jamil
Enter name # 4: huzaifa
Enter name # 5: arshad
Enter name # 6: umar
Enter name # 7: saleem
Enter name # 8: kamran
```

vubookshoppk.com

```
Enter name # 9: babar
Enter name # 10: wasim

ahmed
ali
jamil
huzaifa
arshad
umar
saleem
kamran
babar
wasim
```

This program declares a global array called *pool* that can store all the *Name* objects expected. There is also an associated integer array called *inuse*, which contains *true/false* flags that indicate whether the corresponding entry in the *pool* is in use.
When the statement *directory[i] = new Name(name)* is executed, the compiler calls the class's *new* operator. The *new* operator finds an unused entry in *pool*, marks it as used, and returns its address. Then the compiler calls *Name*'s constructor, which uses that memory and initializes it with a character string. Finally, a pointer to the resulting object is assigned to an entry in directory.
When the statement *delete directory[i]* is executed, the compiler calls *Name* 's destructor. In this example, the destructor does nothing; it is defined only as a placeholder. Then the compiler calls the class's *delete* operator. The *delete* operator finds the specified object's location in the array and marks it as unused, so the space is available for subsequent allocations.
Note that *new* is called before the constructor, and that *delete* is called after the destructor.

vubookshoppk.com

Overloading [ ] Operator to Create Arrays

We know that if we overload operators *new* and *delete* for a class, those overloaded operators are called whenever we create an object of that class. However, when we create an array of those class objects, the global *operator new( )* is called to allocate enough storage for the array all at once, and the global *operator delete( )* is called to release that storage.
We can control the allocation of arrays of objects by overloading the special array versions of *operator new[ ]* and *operator delete[ ]* for the class.

Previously, while employing global *new* operator to create an array of objects, we used to tell the *delete* operator by using the array operator(*[]*) to deallocate memory for an array. But it is our responsibility to provide or to overload different type of *new* and different type of *delete*.

There is a common problem when working with arrays. While traversing elements from the array, we might run off the end of the array. This problem might not be caught by the compiler. However, some latest compilers might be able to detect this.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
int   iarray [10] ;
for ( int  i = 0;   i  <  100;   i++ )
{
        // Some code to manipulate array elements
}
```

If a variable is used in the condition of the loop instead of the constant value, the probability of that error increases. Variable might have an entirely different value than that anticipated by the programmer.

We can overcome this problem of array bound by overloading array operator '*[]*'. As usual before overloading, we should be clear about the functionality or semantics of the array operator. We use array operator to access an element of array. For example, when we write *iarray[5]*, we are accessing the 6th element inside array *iarray*. As we want to check for validity of index every time, an array element is accessed. We can do this by declaring the size of the array using *#define* and checking the *index* against the size every time the array is accessed.

```
#define   MAXNUM   1000
int   iarray [MAXNUM];
```

Below is the syntax of declaration line of overloaded array operator:

```
int& operator   [] ( int   index ) ;
```

In the body of this operator, we can check whether the *index* is greater or equal to the *MAXNUM* constant. If this is the case, the function may throw an exception. At the moment, the function only displays an error message. If *index* is less than *MAXNUM* and greater than or equal to zero, a reference to the value at the *index* location is returned.

Let's write a class *IntArray* and see the array manipulation.

```
/*
The following example defines the IntArray class, where each object contains
an array of integers. This class overloads the [] operator to perform
range checking.
*/

#include   <iostream.h>
#include   <string.h>

class   IntArray
{
public:
    IntArray( int len );
    int   getLength( ) const;
    int &   operator[] ( int index );
    ~IntArray( );
private:
    int   length;
    int   *aray;
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
};


// ------------ Constructor
IntArray :: IntArray( int len )
{
   if( len  >  0 )
   {
      length  =  len;
      aray  =  new  int[len];
      // initialize contents of array to zero
      memset( aray, 0, sizeof( int ) * len );
   }
   else
   {
      length  =  0;
      aray  =  0;
   }
}

// ------------ Function to return length
inline  int  IntArray :: getLength()  const
{
   return  length;
}
// ------------ Overloaded subscript operator
// Returns a reference
int  & IntArray :: operator []( int index )
{
   static  int  dummy  =  0;
   if( (index  >=  0)  &&
      (index  <  length) )
      return  aray[index];
   else
   {
      cout  <<  "Error: index out of range.\n";
      return  dummy;
   }
}

// ------------ Destructor
IntArray :: ~IntArray()
{
   delete  aray;
}
void  main()
{
   IntArray  numbers( 10 );
   int  i;
   for( i = 0; i < 10; i ++ )
```

vubookshoppk.com

```
    numbers[i]  =  i;        // Use numbers[i] as lvalue
  for( i  =  0; i  <  10; i++ )
    cout  <<  numbers[i]  <<  '\n';
}
```

This program first declares *numbers* of type *IntArray* object that can hold ten integers. Later, it assigns a value to each element in the array. Note that the array expression appears on the left side of the assignment. This is legal as the *operator[]* function returns a reference to an integer. This means the expression *numbers[i]* acts as an alias for an element in the private array and it can be the recipient of an assignment statement. In this situation, returning a reference is not simply more efficient but also necessary.

The *operator[]* function checks whether the specified *index* value is within range or not. If it is within the range, the function returns a reference to the corresponding element in the private array. If it is not, the function prints out an error message and returns a reference to a *static* integer. This prevents out-of-range array references from overwriting other regions of memory while causing unexpected program behavior.

vubookshoppk.com

Tips

- The *default constructor* is defined by the C++ compiler automatically for every class that has no default constructor (parameterless constructor) defined already.
- The default constructor (parameterless constructor) is called for each element in the array allocated with *new*.
- The *new* operator returns a *void* *, accepts a parameter of type *size_t*.
- The *delete* operator returns nothing (*void*) and accepts a pointer of *void* * to the memory block.
- With *new* operator function, a block of memory is allocated first and then constructor is called.
- With *delete* operator, destructor of the object is called first and then memory block is deallocated.
- By overloading *new* and *delete* operators, only allocation and deallocation part can be overridden.
- The same pointer that is returned by the *new* operator, is passed as an argument to the *delete* operator. These rules apply to both, if operators (*new* and *delete*) are overloaded as member or non-member operators (as global operators).

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

- By overloading the array operator ([]), one can implement mechanism to check for array bound.

vubookshoppk.com

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Lecture No. 35

Reading Material

Deitel & Deitel - C++ How to Program Chapter. 11

11.1, 11.2, 11.3, 11.3.1, 11.3.2,
11.4, 11.4.1, 11.4.2, 11.4.3, 11.5

Summary

30) Streams
31) Source and Destination of streams:
32) Formatted Input and Output
33) Recap Streams
34) Buffered Input/Output
35) Methods with streams
36) Examples using streams

vubookshoppk.com

Streams

We have been discussing the concept of 'Streams' from the very beginning of the course. In this lecture, various aspects of this concept will be discussed. There are two types of streams i.e. input streams and output streams. Before going into minute details, we will see what actually these two types are. You are well aware of the terms 'cin' and 'cout', used several times in our previous programs. We have used *cout* for output and *cin* for input. Similarly, the terms of file input and file output are very much known to us. We leaned how to write in files and how to read from files. These are also streams. Let's have a look on the things and functions we can do with streams. There are some specific functions for input and output in C. *printf* and *scanf* are normally used in C. In these functions, we have to tell what type of data we are using and in which variable. Streams are counterpart of this in C++. The input output system in C++ is streams. As the name applies, it's a stream of bytes. As told earlier, it may also be termed as a door through which program can communicate with the outside world. For entering some data, we use *cin* stream while the data is read from the keyboard and stored in some variable. To display data on the screen, we can take help of *cout*.

For making the things more comprehensive, we will consider the example of a class-room. Suppose you are sitting in the class and listening to this lecture. All of a

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

sudden, the students are asked by the instructor to go out of the classroom. How would you do it? You will get up from the seat and walk through the door. But you can see that all of you cannot go through the door simultaneously. You would go through the door one by one. Similarly if you have to come to the classroom, you will enter one by one and sit on the seats. In a way you are forming a sequence of people in this case. Stream is a sequence of bytes. It is an ordered sequence. Let's compare it with the door example. The person who enters first will go out of the door first. The person who enters behind someone will go out behind that person. Similarly streams are also ordered sequence. The thing that enters first into the stream will go out first. You should think streams as ordered sequence of bytes. Byte is a unit of measure. A byte can store one character, so you can think of an ordered sequence of characters.

As programmers, we communicate with our programs in English through the keyboard. We may be typing the letters, abc or the numbers, 012 on the keyboard. These all are the characters. In the programs, we store these characters in variables of different data types. Sometimes, these may be in some of our objects. On the keyboard, we type the character 'a' that is stored in some variable *c* in our program. How these two are linked? This link is formed in *cin* stream. Consider *cin* as a pipe or a door. The character 'a' is entered from one side and then a conversion takes place i.e. character is converted into its binary representation, stored in the character variable named *c*. So there is an implicit conversion happening. The same thing happens, if we have an integer *i* and press the key 1 from the keyboard. The digit 1 travels as a character but inside it is stored as number. You have to be careful while dealing with this concept. We have talked about the ASCII characters. On the keyboard, we have alphabets, numbers and symbols. When you press the number key from the keyboard, it goes at number inside the computer. In general terms, it is a character. It means that when you enter a key on your keyboard, a character sequence is generated that later goes into the computer. This character sequence has some binary representation. It does not mean a sequence of characters, but a code that goes inside the computer on pressing a key on the keyboard. This code is called as ASCII code. It is the binary representation of a character. Here both the characters 'A' and 'B' have some binary representation. Similarly '0','1','2' have some binary representation. It does not mean that the character representation of '1' is also '1'. If you are aware of the ASCII table (you have already written a program to display the ASCII table), it will be evident that the binary representation of character '1' is some other value. Similarly all the numbers 1, 2, 3 etc have some ASCII value. So whenever you press any key on the keyboard, its ASCII code goes inside the computer.

Now when we use *cin* stream to read some number from the keyboard and store it in the integer variable, its binary representation is ignored and the value is stored. So *cin* is performing this transformation operation. It is taking a character ASCII code and knows that it is supposed to represent some number. It has the ability to convert it into the appropriate number before putting it into the integer variable *i*. What happen if we use *cin* to read some value and store it in some integer variable and press some alphabet key instead of numeric keys. Some error will occur. So in the *cin* stream, an error can be detected.

Let's us look at the collection of input and output classes. These are objects having functions. This collection of classes and their functions are known as input-output

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

streams in the C++. The most common things that we have been using are *cin*, used to get input from the keyboard and *cout* employed to display something on the screen. So one is input stream and the other one is output stream. Since we are not going to discuss the object- oriented programming in this course. We will not discuss the hierarchy through which these classes are derived. We have some objects for input stream and out put stream e.g. *cin* and *cout* respectively. Being objects, they have member methods that we can call. They also have the operators like '<<' and '>>', as used in the earlier programs. These operators '<<', '>>' are heavily overloaded. What does this mean? If we write *cin >> i;* here *i* is an integer. Automatically *cin* will take a character in ASCII code from the keyboard, convert it into number and store it into the integer variable. On the other hand, if we write *cin >> c;* where *c* is a character data type. When we press a key from the keyboard, it will be stored in *c* as a character. So the stream extractor operator of cin (i.e. >>, which gets the data from the stream and stores it into the variable) is already overloaded. It knows how to behave with int, char, float etc data type and what sort of conversion is required. In case of float number, we have decimal point, *cin* knows how to treat it besides converting and storing it into a float variable. Similarly if we use any character pointer i.e. string, the same >> operator has the capability of reading strings too. Obviously, one operator can't perform all these functions. It seems that we are using the same operator. Internally, this operator is overloaded. It is not the same for int, char, float, string and so on. But due to the operator overloading, its usage is very simple. We just write *cin >> i;* it works perfectly.

Source and Destination of streams

As earlier said that streams are sort of door or pipe between two things. What are these two things? For every stream, there must be some source and some destination. For *cin*, the source is normally keyboard and the destination can be an ordinary variable i.e. native-data type variable. It could be some area of memory or our own data type, i.e. object for which we have overloaded the operator and so on. So always there is a source and there is a destination.

cout is output stream. It takes the data from the program and presents it in human readable form. It also has some source and destination. The source may be some file, or the region in memory or the processor or a simple variable or our own object of our data type. The destination is normally screen. The destination can be a file, screen, or printer etc. You have used file input and file output so you know how it works. When we talk about area in memory, it may be an array that we read or write. It could also be a character string which is itself an area in the memory.

"Every stream has an associated source and a destination"

Now we will talk about yet another concept i.e. the state of stream. What does it mean? We say that *cin >> i* ; where *i* is an integer. When we give it the input 'a' and press the key 'enter', the stream knows that it is a bad input. So it is capable of signaling and setting its state specifying that some thing not good has been done. So from a program, we can always test whether the state of stream is right or not. We should carry out all kinds of error checking, debugging and error handling while writing programs. We don't want to manipulate bad data. So checking for this everywhere will be good. For example, if we write a simple program that takes two

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

integers from key-board, divides one number by the other and displays the result. Following will be the code segment.

```
int i, j ;
cin >> i ;
cin >> j ;
cout << i / j ;
```

Now we have to see what happens if the user gives a value 0 for j. We don't want to divide it by zero as even the computer does not know how to do it. When we have zero in j, our program probably will work through an exception or error before coming to a halt. If we trap this error inside the program, it would be much nicer. We can say if j is not zero, it will be good to carry out the division. So error checking and handling is always important. The same thing applies to I/O streams. When we execute input or output operation, we should check whether the operation has been carried out correctly or not. To do this, we can check the state of the stream.

Here is a simple example showing the simple use of streams.

```
/* Avoiding a precedence problem between the stream-insertion operator and the
conditional operator. */

#include<iostream>

int main()
{
        int x,y;
        cout<< "Enter two integers: ";
        cin>>x>>y;
        cout<<x << (x ==y ? " is" : " is not") <<" equal to "<< y;
        return 0;
 }
```

vubookshoppk.com

The output of the program:

```
Enter two integers: 3  3
3 is equal to 3
```

Formatted Input and Output

Other things that the streams provide us are a capability of formatted input and output. We have been using *cin* and *cout* very simply without any formatting. As the output of a program, we do not want that numbers should be printed in a way that makes it difficult to read and understand. We want to format the output in a way that the numbers are placed correctly at the correct position on the paper. You might have seen the electricity bills or telephone bills printed by the computers. First the empty bills are printed in the printing press containing headings with blank boxes to put in the bill entries. Then the computer prints the entries in these boxes from the system. These entries (data regarding the bill) are printed at correct places on the bill. Sometimes, you see that the entries are properly printed. That is not a computer-fault

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

but only due to poor paper adjustment in the printer. The printing of these entries is carried out by with the use of formatted output.

The second example is the display of a matrix on the screen. Suppose, we want that the numbers of a column are displayed up and down in a column. Similarly the second column should be displayed and so on. At first, we will format these things.

When we do word processing, we type a paragraph. The lines of the paragraph are left justified but ragged on right hand side. In word processing, we have a choice to justify the paragraph. By doing this, the left and right margins of the paragraph are put in a straight line while adjusting the space between the words. Now look what happens if we want to print a string with *cout* and want it left or right justified. This is what we call formatting the output. Similarly, we want to print the value of *pi* which is stored in a variable as 3.1415926. But we want that it should be printed as 3.141 that means up to three decimal places.
There should be a method of formatting it. Thus by formatting the output, the presented representation (which we read as human being) can be different from the internal representation. So we can do a lot of formatting with these streams.

Besides, there are member functions with the streams. Let's look at the member functions of *cin*. The first one is the *get* function. We can use it by writing:

cin.get();

The notation explains that *cin* is an object (cin is an object of input stream) and *get* is a member function of it. This function reads a character. In this case, it reads a single character from key board and returns it as *cin* is calling it. We have two variants of this get function with *cin*. One is that *cin.get* returns a character. We can write it as under:

c = cin.get() ;

The second method is *cin.get(character variable) i.e.* one character at a time. It works with characters, not through number or string. It is one character at a time.

The second function of *cin* is the *read* function. This function differs from the *get* function in the way that it returns a buffer instead of a single character. So we can point to a buffer and tell the number of characters to be read. We normally provide a delimiter, a specific character up to which we want to read. Normally we use the new line character as a delimiter and read a single line at a time.

Thus, we have three ways of obtaining input with *cin,* which is an object of type input stream (istream). If we create an object of type *istream*, it will also get these functions as it is derived from the same class.

We have seen that there are many methods and operators associated with *cin*. The same thing applies to *cout*. *cout* is the output stream which usually, presents the data from the computer in human readable form. The operator associated with *cout* is the stream insertion (<<). That means we insert the operator in the stream and the stream displays the output on the screen. So the operator with cout is << and it displays the

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

value of the data variable that we provide it after the << sign. Thus to display the value of an integer variable i, we can write *cout << i* ; If we want to format the output, it can also be done here.

The cout has a function *write* with it. This function can be used if we want to write a chunk of data from the buffer. Similarly, to output a single character, *cout* has the function named *put*. It can be written as:

cout.put(character variable) ;

Here, it will display the value of the character variable.

Recap streams

Streams are nothing but an ordered sequence of bytes.
They allow data to move from one part of the computer to another which may be the screen or key board from and to, or from memory or files on disc and so on.
Byte stream is used to connect the source and the destination.
These byte streams are implemented as objects. Being objects, they do have their member functions and have their member operators. The member operators are heavily overloaded to allow these streams to handle a variety of data types.
The streams have a state that can be checked by us. We have used eof (end of file) with the file reading. This is a way to check the state of the stream.

While using I/O streams, we have to include some header files. Whenever we use *cin* and *cout*, the file *iostream.h*, is included in which all these classes and objects have been defined. For the formatted input and output, we manipulate the streams. To do stream manipulations, we have to include a header file having the name *iomanip.h*. We can understand that *iomanip* is a short hand for input output manipulation.

Now let's take a look at the standard streams which are provided to our programs. Whenever we write a C++ program and include *iostream.h* in it, we get a stream for input (reading) that is *cin.* This is a built in thing. We can use this object. Similarly, for output (writing), we get *cout*. Other than these, we get some other streams by default. These include *cerr* (read as c error) and clog. To understand these streams, we have to talk about buffered input and output.

Buffered Input/Output

In computers, most of the components relate to electronics like chips, memory, micro processor etc. There are also electro-mechanical things like disc. The key board itself is an electro mechanical accessory. The electro mechanical parts of the computer are normally very slow as compared to the electronic components. So there is a difference between the two in terms of speed. Secondly, every input/output operation costs computer time. Input/output costs and the I/O devices (keyboard, monitor and disc etc) are slower as compared to the speed of the microprocessor and the memory being used. To overcome this speed difference, we use the mechanism, called buffered input/output. Suppose, we have a program which executes a loop. That loop outputs a number in each of iteration to store in a file on the disc. If we write the output number

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

to the disc in each iteration, it will be the horrendously wastage of computer time. It means that the disc is electro mechanical device. Similarly in each iteration, the mechanical movement takes time. But if we gather the data and write it to the disc, there will be one mechanical movement. The heads of the disc will move mechanically once to a point and the whole chunk of data will be written on the disc. This is the more efficient way of using the disc. So whenever we have a program that writes the output data to the disc, it will be nice to collect the output data (numbers) and write it on the disc in one write operation instead of writing the numbers one by one. The area where we gather the numbers is known as buffer. The example stated in this case is the buffered output. In this case, the output does not go directly to the disc. We first gather the data in a buffer and then write it on the disc.

Now think about another situation. Suppose we have a program that performs very complex calculations. It means that there is a *while* loop that performs so heavy calculations that each of the iteration takes, say one minute and then provides the result. Now we want to write that output to a file on the disc and see the iteration number of the loop on the screen. We do not want to write the output to the disc after each iteration. We gather the data in a buffer. In the meantime, we want to see the loop counter on the screen. If we gather this output of counter number in a buffer, it may happen that the buffer gathers the iteration numbers for 250 iterations before displaying it on the screen. Thus, we see on the screen the numbers 1, 2, 3 …..250, when 250 iterations have been performed. There are again 250 numbers gathered at one time. We see numbers 251, 252 ……500, when 500 iterations have been performed. When we start the program, there will be two buffers gathering data. One buffer gathers the data to write to the disc and the other gets the data of iteration numbers to display on the screen. As we said that each iteration takes one minute, meaning that the iteration numbers will not be seen for a long time. Rather, these will be shown after 250 iterations (i.e. 250 minutes). During this period, we do not show any thing on the screen. Here, we are not sure whether the program is executing properly as we do not see any iteration number on the screen. We want to know after every minute that loop has executed once. So after every minute, the loop counter should be displayed on the screen, so that at any time we could see how many iterations have been performed. For this, we need unbufffered output on the screen. Thus, in the same program we require buffered and unbuffered output.

Now these requirements are contradictory with different issues. These are met in our system with the *cerr* object. *cout* is a buffered output. We cannot see it as nowadays compilers are very intelligent. If you know UNIX, or command prompt and input output redirection, we can actually see this in operation. Perhaps, you can create an example to understand this. For the moment just try to understand that *cout* is buffered output. It gathers data and sends it to the screen. On the other hand, *cerr* is unbuffered output. It will show the data on the screen at the same time when it gets it. So *cerr* is an output stream, an *ostream* object but unbuffered. It shows data immediately. So we can use something like the *cerr* object that will show how many loops have been executed. It will use something like *cout* to write on the disk to buffer the output. In this case, we are getting efficiency in addition to information. Normally we use *cerr* object in C++ for this purpose. Besides, we also have *clog*. It is also known as standard log having detailed information of the log. To collect information of the program, we write it with *clog*. Normally when we execute our programs- *cout, cerr* and *clog*, all are connected to screen. We have ways to direct them at different

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

destinations. That depends on the operating system. Suppose, we specify the buffer size, normally the operating system or compiler does this for us. A typical size of buffer is 512 bytes. When the information is of 512 byte size, output will take place. But in the program, we may want at some point that whatever is in the buffer, show them. Is there a way of doing that? The normal mechanism is flush. Flush the stream. The flush command forces the data from the buffer to go to its destination which is normally a screen or file and make the buffer empty.

Uptil now, we have been using two things to end the line. One is new line character i.e. "\n". When we are displaying something on the screen, it makes the next output to start from the next line. The cursor moves to the left margin of the next line on the screen. New line is just a character. The other one was *endl*. If we write *cout << endl;* It seems that the same thing happens i.e. the cursor moves to the left margin of the new line. But *endl* actually does something else. It flushes the output too. As a result, it seems that *cout* is unbuffered i.e. its output is immediately available on the screen. Depending on the compiler and operating system, you may or may not see the buffered effect. But one thing is clear that while reading the source code, you will know where *cout* is used and where *cerr*. Typically, that is also the programming style where the output of *cerr* is informative and small. It shows that where is the control in the program. The output of *cout* is more detailed and the actual output of the program. There are benefits of these things in the code. In case of *cin*, it is alone. For output, we have *cout*, *cerr* and *clog*. In DOS, we have two more output streams i.e. *caux* (auxiliary input output stream) and *cprn* (printer output). These are no more relevant now.

Predefined Stream Objects:

Object	**Meaning**
cin	Standard input
cout	Standard output
cerr	Standard error with unbuffered output.
clog	Standard error with buffered output
caux	Auxiliary (DOS only)
cprn	Printer(DOS only)

Now let's take a look at the operators associated with these streams. We have been using the stream insertion operators '<<' with *cout*. We need to understand that how these operators are implemented. Using *cout*, we can chained the output. It means that we can write as *cout << "The value of the first integer is " << i;* This is the single *cout* statement. How does that work? What happens is the first part goes to *cout*. In this case, it is the string "The value of the first integer is". The data travels in the direction of the arrows. This string is inserted in the stream and displayed on the screen. What about the rest of the statement i.e. *<< i;* this again needs *cout* on the left side to be executed. It should look like as *cout << i;* Once, we have defined that this is the behavior expected by us. Then, we understand that this is exactly the way it has been programmed. The stream insertion operator '<<' is overloaded for the output stream and it returns the reference of the output stream. The syntax of stream insertion operator is:

```
ostream& ostream::operator << (char *text);
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

The important thing to note is that this operator returns the reference to the *ostream* object itself. Whenever we write a chained output statement, it is executed from left to right. So *cout << " The value of the first integer is"* is processed first from left to right. The process is that this character string is displayed on the screen. As per the prototype and definition, it returns the reference to the *ostream* object. In this case, the object was *cout*, so a reference to the *cout* object is returned. Now the rest of the statement becomes as *cout << i;* It is processed quite nicely. This allows the stream insertions to be chained. It is also applicable to the input. So if we say something like *cin >> i >> j;* both *i* and *j* are integers. Now again, it is processed from left to right. This is the *istream* and the extraction operator will return the reference to the *istream* object i.e. *cin*. So at first, the *cin >> i* is processed which will return the reference to the *cin* object. The rest of the statement seems as *cin << j;* It is important to understand how these operators work. You can see their prototypes that they return *iostream* objects themselves. That is the why, we can chain them. Now let's see what are the other various methods associated with these input output streams.

Methods of streams

There are some other functions associated with *cin* stream. We have used some of them. We have used *get()* and *read()* methods with input stream. Another member function of *cin* is *getline()*. It reads a complete buffer i.e. the number of character specified up to a delimiter we specify. We can write something like:

```
cin.getline(char *buffer, int buff_size, char delimiter = '\n')
```

The character data is stored in **buffer*. *buff_size* represents the number of characters to be read. If we specify its value 100, then *getline* will read 99 characters from the keyboard and insert a null character in the end. As you know, in C++ every character string ends with a null character. We can also give it a delimiter. Sometimes, we may want to read less character. Normally, the delimiter is the new line character. So while typing on the keyboard, if we press the enter key then it should stop reading further and put the data into the variable *buffer*. So there is a *getline* fiunction.

There are some other interesting functions also. When we use *cin.get()*, a character is read. We can throw back the character gotten by this *get* function by using the *unget()* function. So we can use *cin.unget() that* will return the most recently (last) gotten single character.

We have a function *peek()*, also written as *cin.peek()*; The purpose of this function is that we can see the next character that we are going to get. This function returns the next character that would be read if we issue *cin.get()*.

All these functions (getline, get, read, unget and peek) are implemented as member functions of the input class.

Similarly, there are functions associated with *cout*. We have *cout.putline()*; which outputs a buffer. Actually we have no need of this function because *cout*, itself, knows how to handle character strings. Then we have *cout.write()*; which can perform a raw, unformatted output. The function *cout.put()*; is like a formatted output. It performs character by character output. We can do many formatting conversions by using the stream insertion operator (i.e. <<) with *cout*. We can write an overloaded function of

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

stream insertion (<<) to input or output a complex number. We know that a complex number has two parts i.e. real and imaginary. We can write the overloaded function such that if we give two numbers with space between them it could read it. We can also write it as that it could read two numbers (that are real and imaginary parts of a complex number) separated by comma. Thus there may be different ways to write the overloaded operator.

The white space character is very significant. We can show it by a simple example. Suppose we have an array *name* of 60 characters. We get a name from the user in this array by using *cin* and then display this string by *cout*. The code segment for this purpose can be written as:

```
char name [60] ;
cin >> name ;
cout << name ;
```

Now when the user enters the name, suppose it enters 'naveed malik' that is a name containing two words with a space between them. When we display this name by using *cout*, only 'naveed' is displayed on the screen. It means that only one word 'naveed' was stored in the array. The reason for it that the streams (cin, cout) are sensitive to white space character that is treated as a delimiter. Now where is the second word 'malik'. It has not got deleted yet. It is in the buffer of the stream. This example will read like the following:

```
char nam1 [30], name2 [30] ;
cin >> name1 >> name2 ;
```

Thus, we have two character arrays now. We can write 'naveed malik' and press enter. The first part before space (naveed) will go to the first array *name1* when that array is used with *cin*. We will write another *cin* with *name2* and the second part (malik) will go to the second array *name2*. So things don't disappear. They stay in the buffer till you actually expect them. We have to be careful about that.

Examples using streams

A simple example showing the use of getline function.

```
// A simple example showing the use of getline function.
#include <iostream.h>

int main()
{
   const int SIZE = 80;
   char buffer[SIZE];

   cout << " \n Enter a sentence: \n" ;
   cin.getline(buffer, SIZE);

   cout << " The sentence entered is: \n" << buffer << endl;
   return 0;
}
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Output of the program.

```
Enter a sentence:
this is a test
The sentence entered is:
this is a test
```

A simple example showing the use of read and write functions.

```
// A simple example showing the use of read and write functions.
#include <iostream.h>

int main()
{
   const int SIZE = 80;
   char buffer[SIZE];

   cout << " \n Enter a sentence: \n" ;
   cin.read(buffer, 20);

   cout << " The sentence entered was: \n";
   cout.write(buffer, cin.gcount());
   cout << endl;
   return 0;
}
```

Output of the program.

```
Enter a sentence:
This is a sample program using read and write functions
The sentence entered was:
This is a sample pro
```

vubookshoppk.com

Lecture No. 36

Reading Material

Deitel & Deitel - C++ How to Program Chapter. 11

11.6, 11.7

Summary

37) Stream Manipulations
38) Manipulators
39) Non Parameterized Manipulators
40) Parameterized Manipulators
41) Format State Slags
42) Formatting Manipulation
43) Showing the Base
44) Scientific representation
45) Exercise

vubookshoppk.com

Stream Manipulations

After having a thorough look into the properties and object definition of I/O streams, we will discuss their manipulations. Here, there is need of some header files to include in our program the way, *cin* and *cout* are used. We include *iostream.h* and *fstream.h* while using file manipulations. In case of manipulation of the I/O streams, the header file with the name of *iomanip.h* is included. It is required to be included whenever there is need of employing manipulators.

As discussed earlier, we can determine the state of a stream. The states of the stream can be determined. For example, in case of *cin,* we can check where the end of file comes.

For state- checking, these stream objects have set of flags inside them. These flags can be considered as an integer or long integer. The bit position of these integers specifies some specific state. There is a bit for the end of file to test. It can be written as under:

```
cin.eof() ;
```

It will return the state of end of file. The bit will be set if the file comes to an end. Similarly, there is a *fail* bit. This bit determines whether an operation has failed or not.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

For example, an operation could be failed due to a formatting error. The statement *cin.fail*; will return the value of the *fail* bit. As this statement returns a value, we can use it in an 'if statement' and can recover the error. Then there is a *bad* bit. This bit states that data has lost. The presence of this bit means that some data has lost during I/O operation. So we can check it in the following manner.

cin.bad();

It can be used in parentheses as a function call that will allow us to check whether the operation failed or successful. Similarly we can also check for' *good',* that is a bit showing that everything is good. This bit will be set if *fail* and *bad* bits are not set. We can check this bit as *cin.good* ; and can find out whether the input operation was successful. If some bit like *bad* has been set, there should also be a mechanism to clear it. For this, we have

cin.clear() ;

as a member function for these objects. This will reset the bits to their normal good state. This is a part of checking input stream.

Manipulators

Whenever carrying out some formatting, we will want that the streams can manipulate and a number should be displayed in a particular format. We have stream manipulators for doing this. The manipulators are like something that can be inserted into stream, effecting a change in the behavior. For example, if we have a floating point number, say pi (л), and have written it as *float pi = 3.1415926* ; Mow there is need of printing the value of pi up to two decimal places i.e. 3.14 . This is a formatting functionality. For this, we have a *manipulator* that tells about width and number of decimal points of a number being printed. Some manipulators are parameter less. We simply use the name of the manipulator that works. For example, we have been using *endl*, which is actually a manipulator, not data. When we write *cout << endl* ; a new line is output besides flushing the buffer. Actually, it manipulates the output stream. Similarly *flush* was a manipulator for which we could write *cout << flush* that means flushing the output buffer. So it manipulates the output.

A second type of manipulators takes some argument. It can be described with the help of an example. Suppose we want to print the value of pi up to two decimal places. For this purpose, there should be some method so that we can provide the number i.e. two (2) up to which we want the decimal places. This is sent as a parameter in the manipulators. Thus we have the parameterized manipulators.

Let's have a look on what streams do for us. We know that streams are like ordered sequence of bytes and connect two things i.e., a source and a destination. In the middle, the stream does some conversion. So it may take some binary representation of some information and convert it into human readable characters. It may also take characters and convert them into an internal representation of data. With in the conversion of data, we can do some other things. For example, you might have seen that if a system prints computerized cheques, it puts some special characters with the numbers. If there is a cheque for four thousand rupees, this amount would be written on it as ****4000.00. The idea of printing * before the amount is that no body could

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

insert some number before the actual amount to change it. As we don't want that somebody has increased the amount on the cheque from Rs 4000 to Rs 14000.The printing of * before the amount is a manipulation that we can do with input or output objects. We can also tell the width for a number to be printed. So there are many conversions that we can do. We can use fill characters like * as mentioned in the example of cheque printing. To accomplish all these tasks, there are different methods. So it becomes a little confusing that the same work is being done through 2-3 different methods. Some are inline manipulators like *endl*. We can use it with << and write inline as *cout << endl* ; The same work could be done with the *flush* method. We could also write *cout.flush* ; Thus it is confusing that there is an inline manipulator and a function for the same work.

Non-Parameterized Manipulators

Let's start with simple manipulators. We have been dealing with numbers like integers, floats etc for input and out put. We know that our number representations are associated with some base. In daily life, the numbers of base 10 are used in arithmetic. When we see 4000 written on a cheque, we understand that it is four thousands written in the decimal number system (base 10). But in the computer world, many systems are used for number representation that includes binary (base 2), octal (base 8), decimal (base 10) and hexadecimal (base 16) systems. A simple justification for the use of these different systems is that computers internally run on bits and bytes. A byte consists of eight bits. Now if we look at the values that can be in eight bits. 256 values (from 0 to 255) can be stored in eight bits. Now consider four bits and think what is the highest number that we can store in four bits. We know that the highest value in a particular number of bits can be determined by the formula 2^n - 1 (where n is the number of bits). So the highest value that can be stored in four bits will be 2^4 - 1 i.e. 15. Thus the highest value, we can store in four bits is 15 but the number of different values that can be stored will be 2^n i.e. 16 including zero. Thus we see that while taking half of a byte i.e. four bits, 16 (which is the base of hexadecimal system) different numbers can be stored in these four bits. It means that there is some relationship between the numbers that have a base of some power of two. So they can easily be manipulated as bit format. Thus four bits are hex. What about eight (octal)? If we have three bits, then it is $2^3 = 8$, which is the base of octal system. Thus, we can use three bits for octal arithmetic.

We can use manipulators to convert these numbers from one base to the other. The manipulators used for this purpose, can be used with *cin* and *cout*. These are non-parameterized manipulators. So if we say the things like *int i = 10* ; Here *i* has the decimal value 10. We write *cout << i* ; and 10 is being displayed on the screen. If we want to display it in octal form, we can use a manipulator here. If we write

```
cout << oct << i ;
```

it will display the octal value of i (10) which is 12. This manipulator converts the decimal number into an octal number before displaying it. So the octal representation of 10 which is 12, will be displayed on the screen. Similarly if we write

```
cout << hex << i ;
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Here hex stands for the hexadecimal. *hex* is the manipulator that goes into the out put stream before *i* and manipulates the stream by converting the decimal number into a hexadecimal number. As a result, the hexadecimal value of 10 is displayed on the screen. If we have a number in octal or hexadecimal system , it can be converted into a decimal number by putting the *dec* manipulator in the stream like

```
cout << dec << i ;
```

These (oct, hex and dec) are very simple inline manipulators without any argument. There is also a manipulator for white space. The white space has a special meaning. It is a delimiter that separates two numbers (or words). In *cin* and *cout*, the white space acts as a delimiter. If we want that it should not act as a delimiter, it can used as a *ws* manipulator. This manipulators skips white space. This manipulator takes no argument. This *ws* manipulator is sometime useful but not all the times. The following table shows the non-parameterized manipulators and their description.

Manipulator	Domain	Effect
dec	In / Out	Use decimal conversion base
hex	In / Out	Use hexadecimal conversion base
oct	In / Out	Use octal conversion base
endl	Output	Inserts a new line and flush the stream
ends	Output	Terminate a string with NULL
flush	Output	Flush the stream
ws	Input	Skip leading whitespace for the next string extraction only

The base becomes important while doing programming of scientific programs. We may want that there is the hexadecimal presentation of a number. We have discussed the justification of using hexadecimal or octal numbers, which is that they match with bits. Here is another justification for it. Nowadays, computers are just like a box with a button in front of them. A reset button is also included with the main power button. While seeing the pictures or in actual Miniframe and mainframe computers, you will notice that there is a row of switches in front of them. So there are many switches in front of these computers that we manipulate. These switches are normally setting directly the values of registers inside the computer. So you can set the value of register as 101011 etc by switching on and off the switches. We can do that to start a computer or signaling something to computer and so on. There are a lot of switches in front of those computers ranging between 8 to 16. You have to simply remember what is the value to start the computer. Similarly, it will require the reading the combinations of switches from the paper to turn on the computer. This combination tells you which switch should be on and which should be off. As a human being instead of remembering the whole pattern like 10110000111 etc, it could be easy to remember it as 7FF. Here we are dealing with HEX numbers. For the digit 7, we need four bits. In other words, there is need to set four switches. The pattern of 7 is 0111. So we set 7 with this combination. For F, all the four bits should be on as 1111 and so on. Thinking octal and hexadecimals straight away maps to the bits. It takes a little bit of practice to effectively map on the switches. On the other hand, decimal does not map to those bits. What will be its octal number in case of decimal number 52.? You have to calculate this. What is the binary representation of 52? Again you have to

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

calculate. There are a lot of things which you have to calculate. On the hand, if we say 7ABC in case of HEX, we are mapping the number straight away on four bits. In octal system, we map the number on three bits. A is ten so it will be 1010 so you can quickly set the switches. It may not be relevant today. But when you are working with big computers, it will become quite relevant. There are many times when we have to manipulate binary data using mechanical device while thinking in hexadecimal and octal terms. In the language, you have the facility to set the base. You can use *setbase()*, *hex, oct, dec* and *setf*. There are many ways of doing the same thing. Programmers write these languages. Therefore they make this facility available in the language as built in.

Parameterized Manipulators

Suppose we want to print the number 10 within a particular width. Normally the numbers are written right justified. In case of no action on our part, *cout* displays a number left justified and in the space required by the number. If we want that all numbers should be displayed within the same particular width, then the space for the larger number has to be used. Let's say this number is of four digits. Now we want that there should be such a manipulator in the output that prints every number in a space of four digits. We have a manipulator *setw* (a short for set width), it takes as an argument the width in number of spaces. So to print our numbers in four spaces we write

```
cout << setw(4) << number ;
```

When printed, this number gets a space of four digits. And this will be printed in that space with right justification. By employing this mechanism, we can print values in a column (one value below the other) very neat and clean.

Now in the example of printing a cheque, we want that the empty space should be filled with some character. This is required to stop somebody to manipulate the printed figure. To fill the empty space, there is need of manipulator *setfill*. We can write this manipulator with *cout* as the following

```
cout << setfill (character) ;
```

where the *character* is a single character written in single quotes. Usually, in cheque printing, the character * is used to fill the empty spaces. We can use any character for example, 0 or x. The filling character has significance only if we have used *setw* manipulator. Suppose, we are going to print a cheque with amount in 10 spaces. If the amount is not of 10 digits, the empty space will be filled with *. Thus the usage of *setfill* is there where we use *setw* for printing a number in a specific width. So if we want to print an amount in 10 spaces and want to fill the empty spaces with *, it can be written as under.

```
cout << setfill(*) << setw(10) << amount ;
```

Thus the manipulators can also be of cascading nature. The stream insertion operator (<<) is overloaded and every overload of it returns a reference to the *cout* object itself. This means that while working from left to right, first the fill character will be set returning a reference to *cout* . Later, its width will be set to 10 character and return a

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

reference to *cout again*. Finally, the amount will be displayed in the required format. Thus, we have manipulated the stream in two ways. Example of a pipe with two bends can help it understand further. Now whatever figure goes into it, its width and the fill character is set and things are displayed in the space of 10 characters. If we want to print an amount of Rs 4000 on the cheque, it will be printed on the cheque as ******4000. Thus, we have two manipulators, *setw* and *setfill* which are used with *cout*.

Let's further discuss the same example of cheque. In real world, if we look at a computer printed cheque , the amount is printed with a decimal point like 4000.00 even if there is no digit after decimal point. We never see any amount like 4000.123, as all the currencies have two- digit fractional part. Thus, we examine that the fractional part has been restricted to two decimal places. The decimal digits can be restricted to any number. We have a manipulator for this purpose. The manipulator used for this purpose is *setprecision*. This is a parameterized manipulator. It takes an integer number as an argument and restrict the precision to that number. If we write

```
cout << setprecision (2) << float number ;
```

The above statement will display the given float number with two decimal places. If we have the value of pi stored in a variable, say *pi*, of type float with a value of 3.1415926 and want to print this value with two decimal places. Here, manipulator *setprecision* can be used. It can be written as under.

```
cout << setprecision (2) << pi ;
```

This will print the value of *pi* with two decimal places.

Now think about it and write on the discussion board that whether the value of pi is rounded or truncated when we print it with setprecision manipulator. What will be the value of pi with five decimal places and with four decimal places? Will the last digit be rounded or the remaining numbers will be truncated?

At this point, we may come across some confusion. We have learned the inline manipulators that are parameter less. For these, we simply write *cout << hex << number;* which displays the number in hexadecimal form. There is also a parameterized manipulator that performs the same task. This manipulator is *setbase*. It takes the base of the system (base, to which we want to format the number) as an argument. Instead of using *oct, dec* and *hex* manipulators, we can use the *setbase* manipulator with the respective base as an argument. So instead of writing

```
cout << oct << number ;
```

we can write

```
cout << setbase (8) << number ;
```

The above two statements are equivalent in the way for having the same results. It is a matter of style used by one of these manipulators. We can use either one of these, producing a similar effect. The *cout << setbase(8)* means the next number will be printed in the base 8. Similarly *cout << setbase(16)* means the next number will be printed in hexadecimal (base 16) form. Here a point to note is that *setbase (0)* is the same as *setbase(10).*

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Following is the table, in which the parameterized manipulators with their effect are listed.

Manipulator	Domain	Effect
resetioflags(long f)	In / Out	Clear flags specified in f
setbase (int b)	In / Out	Set numeric conversion base to b (b may be 0, 8, 10 or 16)
setfill (int c)	Output	Set fill character to c
setiosflags(long f)	In / Out	St flags specified in f
setprecision (int p)	Output	Set floating point precision to p
setw (int w)	Output	Set field width to w

Format State Flags

We have discussed that there are flags with the stream objects. This set of flags is used to determine the state of the stream. The set includes *good, fail, eof* etc that tells the state of the stream. There is also another set of flags comprising the ones for input/output system (ios). We can use *setioflag,* and give it as an argument a *long* number. Different bit values are set in this number and the flags are set according to this. These flags are known as format state flags and are shown in the following table. These flags can be controlled by the *flags*, *setf* and *unsetf* member functions.

Format state flag	**Description**
ios::skipws	Skip whitespace character on an input stream.
ios::left	Left justify output in a field, padding characters appear to the right if necessary.
ios::right	Right justify output in a field, padding characters appear to the left if necessary.
ios::internal	Indicate that a number's sign should be left justified in a field and a number's magnitude should be right justified in that same field (i.e. padding characters appear between the sign and the number).
ios::dec	Specify that integers should be treated as decimal (base 10) values.
ios::oct	Specify that integers should be treated as octal (base 8) values.
ios::hex	Specify that integers should be treated as hexadecimal (base 16) values.
ios::showbase	Specify that the base of a number is to be output ahead of the number(a leading 0 for octals, a leading 0x or 0X for hexadecimals).
ios::showpoint	Specify that floating-point numbers should be output with a decimal point. This is normally used with ios::fixed.
ios::uppercase	Specify that uppercase letters (i.e X and A through F) should be used in the hexadecimal integers and the uppercase E in scientific notation.
ios::showpos	Specify that positive and negative numbers should be preceded by a + or - sign, respectively.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

ios::scientific	Specify output of a floating-point value in scientific notation.
ios::fixed	Specify output of a floating-point value in fixed point notation with a specific number of digits to the right of the decimal point.

Let's talk about more complicated things. We discussed a parameterized manipulator *setw* that sets the width to print in the output. There is an alternative for it i.e. the member function, called '*width()'*. This function also takes the same parameter and an integer, in which width the things are to display or read. This function applies to both input and output stream. For this, we write *cin.width (7).* This will create a format field of the width of 7 characters for an input. Now we write *cout.width (10)* ; this will set the width of output field to 10. With it, the next number to be printed will be printed in 10 spaces. Thus *setw*, inline manipulator has the alternative function *cin.width* and *cout.width* with single argument.

It equally applies to the *setprecision*. This is the parameterized, inline- manipulator that sets the places after the decimal point. There is a member function as well in these objects that is *precision*. The *setprecision* is an inline manipulator, used along with stream insertion (<<). If we want to do the same thing with a function call, *cout.precision(2)* is written. It has the same effect as that of *cout << setprecision (2).* Thus we have different ways of doing things.
We have used *setfill* manipulator. Here is another member- function i.e. *cout.fill*. The behavior of this function is exactly the same. We simply write *cout.fill('*')* ; identical to *cout << setfill('*')*. The filling character is mostly used whenever we use financial transactions but not necessarily. We can also use zero to fill the space.
So *fill* and *setfill*, *width* and *setw*, *precision* and *setprecision* and almost for every inline manipulator, there are member functions that can be called with these streams.
The member functions are defined in *iostream.h. However,* the manipulators are defined in *iomanip.h*. Normally we have been including *iostream.h* in our programs to utilize the member functions easily. But inclusion of a header file 'iomanip.h file is must for the use of manipulators.
We should keep in mind that when we can write inline manipulators in the following fashion.

```
cout << setw (7) << i ;
```

And in the next line we write

```
cout << j ;
```

Here the *setw* manipulator will apply to *i* only and not after that to *j*. This means that inline manipulators apply only to the very next piece of data i.e. output. It does not apply to subsequent output operations.

Formatting Manipulation

We can adjust the output to left side, right side or in the center. For this purpose, we have a member function of the object whose syntax is as under:

cout.setf(ios:: *flag*, ios:: adjust field)

The *setf* is a short for set flag. The flags are long integers, also the part of the objects. They are the bit positions, representing something. Here we can set these flags. The

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

flags of *adjustfield* are set with values i.e. left, right, left | right and internal. The description of these is as follows.

Value of flag	Meaning	Description
left	Left-justify output	Justifies the output to left side
right	Right-justify output	Justifies the output to right side
left | right	Center output	Centralized the output
internal	Insert padding	Places padding between signs or base indicator and the first digit of a number. This applies only to number values and not to character array.

Following is the code of a program that shows the effects of these manipulators.

```
//This program demonstrate the justified output

#include <iomanip.h>
#include <iostream.h>

void main()
{
        int i = -1234;
        cout.setf(ios::left, ios::adjustfield);
        cout << "|" << setw(12) << i << "|" << endl;
        cout.setf(ios::right, ios::adjustfield);
        cout << "|" << setw(12) << i << "|" << endl;
        cout.setf(ios::internal, ios::adjustfield);
        cout << "|" << setw(12) << i << "|" << endl;
        cout.setf(ios::left | ios::right,
        ios::adjustfield);
        cout << "|" << setw(12) << i << "|" << endl;
        cin >> i ;
}
```

Following is the output of the above program.

```
|-1234       |
|       -1234|
|-       1234|
|       -1234|
```

We have discussed two types of manipulators for base, the parameter less manipulator in which we look for *oct*, *dec* and *hex*. On the other hand, there is a parameterized manipulator *setbase* which takes an integer to set the base. It uses 0 or 10 for decimal, 8 for octal and 16 for hexadecimal notations.
Now we have a generic function *setf* that sets the flags. We can write something like

cout.setf(ios::hex)

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

The hex is defined in *ios*. It has the same effect. It sets the output stream, in this case *cout*, to use hexadecimal display for integers. So it is the third way to accomplish the same task. The use of these ways is a matter of programming style.

Showing the base

Now there should be someway to know which base has the number output by the programmer. Suppose we have a number 7ABC, then it be nothing but hexadecimal. What will be the nature of 7FF. It is hexadecimal. However, the number 77 (seven seven) is a valid number in all of different basis. We have a built-in facility *showbase*. It is a flag. We can set the *showbase* for output stream that will manipulate the number before displaying it. If you have the *showbase* falg on (by default it is off), a number will be displayed with special notations. The *setf* function is used to set the flag for the base field. Its syntax is as under:

cout.setf(ios::*base*, ios::basefield);

Here base has three values i.e. oct, dec and hex for octal, decimal and hexadecimal systems respectively. If the basefield is set to *oct* (octal), it will display the number with a preceding zero. It shows that the number is in octal base. If the basefield is set to *hex* (hexadecimal), the number will be displayed with a preceding notation 0x. The number will be displayed as such if the basefield is set to *dec* (decimal). If there is a number, say 77, it will be difficult to say that it is in octal, decimal or hexadecimal base, a valid number for all the three systems. However, if we output it with the use of *showbase,* it will be easy to understand in which base the output number is being represented. The following example, demonstrates this by showing the number (77) along with the base notation.

```
/* This program demonstrate the use of show base.
It displays a number in hex, oct and decimal form.
*/

#include <iostream.h>

void main()
{
   int x = 77;
   cout.setf(ios::showbase);
   cout.setf(ios::oct,ios::basefield);    //base is 8
   cout << x << '\n';                          //displays number with octal notation
   cout.setf(ios::hex,ios::basefield);    //base is 16
   cout << x << '\n';                          //displays number with hexadecimal notation
   cout.setf(ios::dec,ios::basefield);
   cout << x << '\n';
}
```

Following is the output of the program.

```
 0115
0x4d
77
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Scientific Representation

When the numbers get bigger, it becomes difficult to write and read in digits format. For example, one million will be written as 1000000. Similarly hundred million will be 100000000 (one with eight zeros). How will we display the number which is of 20-digit long? For this, we use scientific notation. To do it, a manipulator *ios:: scientific* can be used. If the flag in *setf* to the *scientific* is set, it can be written as

```
cout.setf(ios::scientific, ios::floatfield) ;
```

Then the floating point numbers will be displayed in scientific notation. A number in scientific is like 1.946000e+009. So we can set the state of output stream to use scientific notation for outputting a number.
To do the scientific notation off and restore the default notation, we set the flag in *setf* function to *fixed* (which is a short for fixed point notation). This can be written as

```
cout.setf(ios::fixed, ios::floatfield) ;
```

Uppercase/Lowercase Control

Similarly, we have a manipulator *ios::uppercase*. While using this manipulator, the e in scientific notation is written in uppercase i.e. E. If we are using hexadecimal numbers, then the characters of it will be displayed in uppercase letters as A, B, C, D, E and F.

vubookshoppk.com

Exercise

- We have been using matrices i.e. a two dimensional array. As an exercise, try to print out a matrix of three rows and three columns, in such a way that it should be nicely formatted. The numbers should be aligned as we write it in a neat and clean way on the paper. You can use the symbol | at the start and end of each row as we don't have a so big square bracket to put around three rows. To be more elegant to print a matrix, we can use a proper graphic symbol to put square brackets around the matrix instead of using | symbol. In the ASCII table, there are many symbols that we can use to print in our programs. We have the integer values of these symbols in the table. Suppose you have a value 135 of a symbol. Now to print this symbol, press the 'alt' key and keeping the key pressed enter the integer value i.e. 135 from the num pad of the key board, release the 'alt' key. Now you will see that symbol on the screen. For the value 135, the symbol is ç. In programming, we can provide this symbol to be printed as a single character in single quotes. For this, put a single quote and then enter the symbol in the way stated above and then put the single quote. It will be written as 'ç'. Find out proper symbols from the ASCII table that can comprise to put a square bracket around the matrix.
- Write simple programs to demonstrate the use of different manipulators and examine their effects.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Lecture No. 37

Reading Material

Deitel & Deitel - C++ How to Program ***Chapter 11***
11.3, 11.3.1, 11.4, 11.4.1

Summary

- Overloading Insertion and Extraction Operators
- Example 1
- Example 2
- Tips

vubookshoppk.com

Overloading Insertion and Extraction Operators

We are already aware that while overloading operators, functions are written and spirit or behavior of the operators (+, -, *, /) is maintained in their implementations. Similarly the operator's spirit is kept intact while overloading stream insertion and extraction operators.

We get an integer as input by writing the following lines:

```
int  i;
cin  >>  i;
```

Have a look on the stream extraction operator's (>>) behavior here. The similar behavior is maintained when we overload this stream extraction operator (>>) or stream insertion operator (<<).

There are couple of important things to take care of, before starting implementation for overloading an operator:

The first thing to see is the type of the operator i.e. whether the operator is binary or unary. The binary operator takes two operands while unary operator takes one. The number of operands for an operator cannot be changed while overloading it.
Secondly, the programmer has to take care of, what an operator is returning back. For example, in case of addition (+), it returns back the result of addition. So the cascading statement like *a + b + c;* can be executed successfully. In this case, at first, *b + c* is executed. The result of this operation is returned by the *operator +*, to add it in the variable *a*. So in actual, the operation is carried out as:

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

a + (b + c).

We want to overload stream extraction (>>) and insertion (<<) operators which are actually already overloaded. See the code lines below:

```
1.    int  i  =  123;
2.    double  d  =  456.12;
3.    float  f  =  789.1;
4.
5.    cout  <<  i  <<  "  ";
6.    cout  <<  d  <<  "  ";
7.    cout  <<  f;
```

You can see the lines 5, 6 and 7. The same stream insertion operator (<<) has been used with different data types of *int*, *double* and *float*. Alternatively, these lines (5, 6 and 7) can be written within statement of one line:

```
cout  <<  i  <<  "  "<<  d  <<  "  "<<  f;
```

Similarly, the stream extraction operator (>>) is used with different data types in the following manner:

```
cin  >>  i;
cin  >>  d;
cin  >>  f;
```

Here, stream extraction operator is used with different data types of *int*, *double* and *float*. The three lines given above can be written in one cascading line:

```
cin  >>  i  >>  d  >>  f;
```

The file *iostream.h* contains the operator overloading declarations for these stream insertion (<<) and extraction (>>) operators for native data types. The declarations inside this file look like the following:

```
istream& operator>>(char*);
istream& operator>>(unsigned char* p)
 {
   return operator>>((char*)p);
 }
istream& operator>>(signed char*p)
 {
    return operator>>((char*)p);
    }
istream& operator>>(char& c);
 istream& operator>>(unsigned char& c)
      {
       return operator>>((char&)c);
      }
```

vubookshoppk.com

```
istream& operator>>(signed char& c)
 {
  return operator>>((char&)c);
   }
istream& operator>>(int&);
istream& operator>>(long&);
#if defined(__GNUC__)
__extension__ istream& operator>>(long long&);
__extension__ istream& operator>>(unsigned long long&);
#endif
istream& operator>>(short&);
istream& operator>>(unsigned int&);
istream& operator>>(unsigned long&);
istream& operator>>(unsigned short&);
#if _G_HAVE_BOOL
istream& operator>>(bool&);
#endif
istream& operator>>(float&);
istream& operator>>(double&);
istream& operator>>(long double&);
istream& operator>>( __manip func)
 {
  (*func)(*this); return *this;
 }
 istream& operator>>(__imanip func)
   {
    return (*func)(*this);
   }
 istream& operator>>(streambuf*);
```

vubookshoppk.com

function returns. Therefore, it does not make sense to return references of the objects, passed by value to the function.

As we are declaring this operator function as *friend* of our class *Vehicle*, the *private* members of *Vehicle* will be accessible to this operator function. For example, *tyre* is a *private* data member of type *int* inside *Vehicle* class and inside the operator function's implementation, we can access it by simply writing *v.tyre* as:

```
output  <<  v.tyre;
```

The above statement actually is:

```
cout  <<  v.tyre;
```

tyre is of native data type *int*. The *output* will work for native data types as it is actually *cout*, which is overloaded for all native data type. We are constructing a building using the basic building blocks. We can use the already used bricks to construct new walls. Similarly, while writing out programs, we implement our overloaded operators using the already available functionality of native data types.

Here is how we overload stream insertion operator (<<) for our *Date* class:

```
#include <iostream.h>
class Date
{
   friend  ostream&  operator  << ( ostream &  os, Date  d );
           // this non-member function is a friend of class date
  . . .
  . . .
};


ostream &  operator  << ( ostream &  os, Date  d )
{
  os  <<  d.day  <<  "."  <<  d.month  <<  "."  <<  d.year;  // access private
      data
                                                          // as friend
  return  os;
};
```

Likewise, we can overload stream extraction operator (>>). All the conditions for overloading this operator are similar to that of stream insertion operator (>>). It cannot be a member operator, always a non-member operator function, declared as *friend* of the class to be overloaded for. It returns an object of type *istream &*, accepts first parameter of type *istream &*. There is one additional restriction on extraction operator (>>) i.e. the second parameter is also passed by reference as that object is modified by this operator function. For our *Date* class, it is declared as:

```
istream &  operator  >>  ( istream &  input, Date &  d );
```

Note that second parameter can also be passed by reference for insertion operator (<<) but that is not mandatory and may be used to gain performance. But in case of extraction operator (>>), it is mandatory to have second parameter of reference type.

vubookshoppk.com

Example 1

Following is our* Date *class containing the overloaded insertion (<<) and extraction (>>) operators:

```
/* Date class containing overloaded insertion and extraction operators. */
# include <iostream.h>

class Date
{
    public:
    Date( )
    {
            cout << "\n Parameterless constructor called ...";
            month  =  day  =  year  =  0;
     }

    ~Date ( )
    {
//          cout << "\n Destructor called ...";
     }

   // Methods, not directly related to the example have been taken out from the class

      friend  ostream &   operator  << ( ostream &  os, Date  d );
      friend  istream &   operator  >> ( istream &  is, Date & d );

     private:
                int   month, day, year;
};

ostream &   operator  << ( ostream &  os, Date  d )
{
      os  <<  d.day  <<  "."  <<  d.month  <<  "."  <<  d.year;  // access private data of
                                                                  //Date being a friend
      return   os;
};

istream &   operator  >> ( istream &  is, Date&  d )
{
  cout   <<   "\n\n Enter day   of the date: ";
  cin   >>   d.day;
  cout   <<   " Enter month of the date: ";
  cin >> d.month;
  cout   <<   " Enter year  of the date: ";
  cin   >>   d.year;
```

vubookshoppk.com

```
  return   is;
};

main(void)
{

   Date date1, date2;
   cout <<  "\n\n Enter two dates";
   cin  >> date1 >> date2;
   cout   <<   "\n Entered date1 is: "   <<   date1 << "\n Entered date2 is: " << date2;

}
```

The output of the program is:

```
Parameterless constructor called ...
Parameterless constructor called ...

Enter two dates: ...

Enter day   of the date: 14
Enter month of the date: 12
Enter year  of the date: 1970

Enter day   of the date: 05
Enter month of the date: 09
Enter year  of the date: 2000

Entered date1 is: 14.12.1970
Entered date2 is: 5.9.2000
```

vubookshoppk.com

Example 2

Following is an example of a *Matrix* class, where until now, we have not overloaded insertion (<<) and extraction operators (>>).

```
/* Matrix class, which is without overloading stream operators */
#include <iostream.h>
#include <stdlib.h>

class Matrix
{
    private :

        int numRows, numCols ;
        float elements [30] [30] ;

    public :
        Matrix( int rows , int cols ) ;
```

```
        void getMatrix ( ) ;
        void displayMatrix ( ) ;

};

Matrix :: Matrix ( int rows = 0 , int cols = 0)
{
  numCols = cols ;
        numRows = rows ;

  for ( int i = 0 ; i < numRows ; i ++ )
  {
          for ( int j = 0 ; j < numCols ; j ++ )
       {
              elements [ i ] [ j ] = 0 ;
       }
  }
}

void Matrix :: getMatrix ( )
{
   for ( int i = 0 ; i < numRows ; i ++ )
   {
      for ( int j = 0 ; j < numCols ; j ++ )
      {
         cin >> elements [ i ] [ j ] ;
      }
   }
}


void Matrix :: displayMatrix ( )
{
   for ( int i = 0 ; i < numRows ; i ++ )
   {
      cout << "|  " ;
      for ( int j = 0 ; j < numCols ; j ++ )
      {
         cout << elements [ i ] [ j ] << "  " ;
      }
      cout << "|" << endl ;
   }
}

void main ( )
{

   Matrix matrix (2, 2) ;

   matrix.getMatrix ( ) ;
```

vubookshoppk.com

```
    matrix.displayMatrix ( ) ;

    system ( "PAUSE" ) ;

}
```

The operator functions (<<, >>) are not overloaded for this program. A specific function *getMatrix()* has been called to get the values for the *matrix* object this entirely a different way than we used to do for primitive data types. For example, we used to get *int* i as; cin >> i. Similarly, we called a method *displayMatrix()* to display the values in the *matrix* object. We can see here, if we overload insertion (<<) and extraction (>>) operators then the user of our class, does not need to know the specific names of the functions to input and display our objects.

The changed program after overloading insertion, extraction operators and few additional statements to format the output properly:

```
/* Matrix class, with overloaded stream insertion and extraction operators. */
#include   <iostream.h>
#include   <stdlib.h>

class   Matrix
{
        float   elements[30][30];
        int   numRows, numCols;

public:
        Matrix ( int rows = 0 , int cols = 0 )
        {
                numRows   =   rows;
                numCols   =   cols;
        }

        friend ostream &   operator   << ( ostream & , Matrix & );
        friend istream &   operator    >> ( istream & , Matrix & );
};

istream &   operator  >> ( istream &   input , Matrix &   m )
{
   for ( int i = 0;  i < m.numRows;  i ++ )
   {
      for ( int j = 0;  j < m.numCols;  j ++ )
                {
                        input   >>   m.elements [ i ] [ j ] ;
                }
        }
        return   input;
}
```

vubookshoppk.com

```
ostream &   operator   << ( ostream &   output , Matrix &   m )
{
        for ( int r = 0;  r  <  m.numRows;  r++ )
        {
                for ( int c = 0;  c < m.numCols;  c++ )
                {
                        output   <<   m.elements [ r ] [ c ]   <<   '\t' ;
                }
                output   <<    endl;
        }
        return   output ;
}

int   main ( )
{

        Matrix   matrix ( 3 ,3 );
        cout   <<   "\nEnter a 3 * 3 matrix \n\n";
        cin   >>   matrix ;
        cout    <<   "\nEntered matrix is: \n";
        cout   <<   matrix;

        system ( "PAUSE" );
       return 0;
}
```

The output of the program is:

```
Enter a 3 * 3 matrix

45
65
34
23
72
135
90
78
45

Entered matrix is:
45      65      34
23      72      135
90      78      45
Press any key to continue . . .
```

You can see both the operators are declared *friend*s of the *Matrix* class so that they can directly access the *private* members of the *Matrix*.
The insertion operator (<<) is accepting both the parameters left and right by reference. We already know that for insertion operator (<<), it is not really required

vubookshoppk.com

to pass the second parameter (the *Matrix* object in this case) by reference but we have used here to gain efficiency. The function is returning an object *ostream &*, i.e., it is returning a reference to a *ostream* object, that actually is the required in order to support cascaded operations using this operator.
The extraction operator (>>) is also accepting both the parameters by reference. But for this operator, it is mandatory to accept the *Matrix* object by reference because this function is modifying that object. Similar to the insertion operation, this function is also returning a reference to *istream* object in order to support cascaded operations.
Clearly after overloading the operators << and >>, it is more convenient for the programmer to use these already familiar operators to display and input the object data members. Readability of the program has also comparatively increased.

Tips

- ***Stream insertion (<<) and extraction operators (>>) are always implemented as non-member functions.***
- **operator** << ***returns a value of type*** **ostream &** ***and*** **operator >>** ***returns a value of type*** **istream &** ***to support cascaded operations.***
- ***The first parameter to*** **operator** << ***is an*** **ostream &** ***object.*** **cout** ***is an example of an*** **ostream** ***object. Similarly first parameter to*** **operator >>** ***is an*** **istream &** ***object.*** **cin** ***is an example of an*** **istream** ***object. These first parameters are always passed by reference. The compiler won't allow you to do otherwise.***
- ***For operator >>, the second parameter must also be passed by reference.***
- The second parameter to *operator* << is an object of the class that we are overloading the operator for. Similar is the case for *operator* >>.

vubookshoppk.com

Lecture No. 38

Reading Material

Deitel & Deitel - C++ How to Program Chapter. 11, 7, 3

11.6.4, page 181, 7.7, 3.10

Summary

46) User Defined Manipulator
47) Examples of user defined manipulator
48) Static keyword
49) Static Objects
50) Static data members of a class

Today, we will discuss the concepts like 'user-defined manipulators' and 'static keywords'. Despite being considered minor subjects, these become very important while carrying out complex programming. Let's start with 'User-defined manipulators'.

User Defined Manipulators

We have talked a lot about the manipulators that are provided with the streams in the C++. These are similar to 'setw' function, used to set the width of the output. These are employed as inline like *cout << endl << i;* Remember that these functions work for only the immediately next output. How can we write our own manipulator? To determine it, it is better to understand what parameter-less manipulators are? These are the manipulators without any parameter like *endl.* This is a parameter-less built-in manipulator that inserts the new line besides flushing the buffer. If we want to write our own manipulator, how can we do this? In case of operator overloading, it is pre-requisite to know that where the operator will be used, what will be on its left-hand and right-hand sides. On reviewing the manipulators, you will find a stream object, normally on the left-hand side. Here, we are talking about *ostream*, an output stream. So that object will be *cout*. The *cout* will take this manipulator to carry out some manipulation. These are written in cascading style as *cout << manipulator << "some data" << end*l. With this cascading style, you can get a hint about the operation of this manipulator and its requirements. The point is, the left-hand side is going to be *ostream* object that will call the manipulator. What will be passed to the manipulator and what will be the return type.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Normally on the right-hand side of the manipulator, we have another stream insertion operator i.e. <<. Here we are considering a parameter-less manipulator, that is no argument or number will be passed to it. It may be something like inserting a tab between two numbers for formatting or a manipulator to end the line or to make a sound of bell and so on. The left hand side is *ostream* object. There are no other parameters. The right-hand side is normally a stream insertion operator. We use it as *cout << manipulator* which is itself an action. We overload the stream insertion operator in such a way that the cascading works. So we return an *ostream* object. More accurately, a reference to *ostream* objects is returned. Manipulator is also going to be used in the same way, so that it returns a reference to an object of type *ostream*. Therefore we want to return the *cout* object or whatever stream we are using. Secondly it also needs the object that is calling it. Here we are not talking about our own class. *ostream* class is built-in and not under our control. So it can not be modified. We can only extend it by defining external things. So it is not a member function or member operator, but only a standalone operator. Normally the declaration of this manipulator is as:

```
ostream& manipulator_name (ostream& os)
```

This is also not a friend function. We cannot define friends for the classes that are already written and not in our control. The argument *os* here is the same object which is calling this function. We have to explicitly declare it. After this, we have to define this. Definition is just as another function. You can always write whatever you want inside the function. But we have to look at the spirit of the manipulator. When we are talking about the spirit of the manipulator, it means that the manipulator should only do something regarding output and return. It is normally very simple. Its return type is *ostream* object. In case of tab character, we can write as return *os << '\t';* It can be bell or something else. We can write useful manipulators to leave single or double blank lines or formatting the strings etc. Remember that it has to return a reference of object of type *ostream*. It automatically gets that object as parameter passed in to the function.

Examples of user defined manipulator

Here is the sample program using the manipulators.

```
/*  A small program which uses the user defined manipulators.
*/

#include <iostream.h>
#include   <stdlib.h>

// Gives System Beep
ostream & bell ( ostream & output )   // Manipulator
{
   return output << '\a' ;
}
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
// Gives Tab
ostream & tab ( ostream & output )     // Manipulator
{
   return output << '\t' ;
}

// Takes the cursor to next line
ostream & endLine ( ostream & output )   // Manipulator
{
   return output << '\n' << flush ;
}

void main ( )
{
    cout << "Virtual " << tab << "University" << bell << endLine ; // Use of Mainpulator
    system ( "PAUSE" ) ;
}
```

Lets see another example of matrix using the user defined manipulators for displaying the matrix on the screen.

Here is the code:

```
/*
A small program showing the use of user defined manipulators.
The display function of matrix is using these manipulators to
format the display.
*/

#include <iostream.h>
#include <stdlib.h>
#include <iomanip.h>

// definition of class matrix
class Matrix
{
    private:
       int numRows;
       int numCols;
       float elements[3][3];

    public:
      // constructor
      Matrix(int rows = 0, int cols = 0)
      {
         numRows = rows ;
         numCols = cols;
      }

      // overloading the extraction and insertion operators
      friend ostream & operator << ( ostream & , Matrix & );
```

vubookshoppk.com

```
        friend istream & operator  >> ( istream & , Matrix & );
        // defining the user defiined manipulators
        friend ostream & spaceFirst ( ostream & );
        friend ostream & spaceBetween ( ostream & );
        friend ostream & line ( ostream & );
        friend ostream & newLine ( ostream & );
        friend ostream & star ( ostream & );
        friend ostream & sound ( ostream & );
};
//defining the operator >>
istream & operator >> ( istream & input , Matrix & m )
{
   for ( int i = 0 ; i < m.numRows ; i ++ )
   {
      for ( int j = 0 ; j < m.numCols ; j ++ )
      {
       input >> m.elements [ i ] [ j ] ;
      }
   }
   return input;
}

//defining the operator <<
ostream & operator << ( ostream & output , Matrix & m )
{
   for ( int i = 0 ; i < 60 ; i ++ )
   {
      if ( i == 30 )
      {
          output << "Displaying The Matrix" ;
      }
      else
      {
          output << star ;
      }
   }
   output << newLine;
   for ( int r = 0 ; r < m.numRows ; r++ )
   {
      output << spaceFirst << line;
      for ( int c = 0 ; c < m.numCols ; c++ )
      {
         output << spaceBetween << m.elements [ r ] [ c ] << sound << spaceBetween ;
      }
      output << spaceBetween << line;
      output << newLine;
   }
   output << newLine;
   return output;
}
```

vubookshoppk.com

```
//defining the user defined manipulator, inserting the space
ostream & spaceFirst ( ostream & output )
{
   output << setw(33);
   return output;
}

//defining the user defined manipulator, inserting the space
ostream & spaceBetween ( ostream & output )
{
   output << setw ( 4 );
   return output;
}
//defining the user defined manipulator, inserting the | sign
ostream & line ( ostream & output )
{
   output << "|" ;
   return output ;
}
//defining the user defined manipulator, inserting the new line
ostream & newLine ( ostream & output )
{
   output << endl;
   return output;
}
//defining the user defined manipulator, inserting the *
ostream & star ( ostream & output )
{
   output << "*" ;
   return output ;
}
//defining the user defined manipulator, making sound
ostream & sound ( ostream & output )
{
   output << "\a" ;
   return output ;
}

// the main function
int main ( )
{
     // declaring a matrix of 3*3, taking its input and displaying on the screen
     Matrix matrix( 3, 3);
     cin >> matrix;
     cout << matrix;
     system("PAUSE");
     return 0;
}
```

The output of the program:

vubookshoppk.com

```
3
5
1
8
7
6
2
5
2
*****************************Displaying   The Matrix*****************************

                    |  3   5   1  |
                    |  8   7   6  |
                    |  2   5   2  |

Press any key to continue . . .
```

Static keyword

We have been using static keyword in our examples. What is the meaning of static? The word refers to something that is stationary, stopped and not moveable. What are the types of these variables, declared as static? How can we make use of them? Static as the word implies are variables which exist for a certain amount of time, much longer than that by ordinary automatic variables. Let's consider the example about the lifetime of data variables. One of the variable types is global variable. Global variables are those that are defined outside of main. They are written as standalone statements before main function as *int i;* the variable *i* is a global variable. It is not only accessible in main but also in all the functions. They can assign some value to *i* or obtain the value of *i*. Global variables come into existence whenever we execute the program and the memory is allocated for *i*. It exists all the time when the program is running. At the end of the program execution, the memory will be de-allocated and returned to the operating system. So it has a very long lifetime. We need a value which exists for the complete execution of the program and is available in all the functions. We use global variables. It is not a good idea to do that unless it is absolutely necessary. The major plus point of these variables is that these are accessible from everywhere in the program. The inconvenience is that theses variables are visible in those functions too which does not need them.

Suppose, we have a global variable *i* declared as *int i;* and in some function we are writing a *for* loop as *for(i = 0; i < n; i++);* Now which *i* is being used here. This is the variable *i*, declared as global. This global *i* may have some valuable value, like the number of cars or the number of students etc. Here, in the function when we run the loop, the value of *i* will be changed. The global variables have this bad habit of being around, even when we don't need them. What will happen if we declare another *i* variable inside the function? A local variable will be created inside the function at run time and the global *i* is not going to be accessible. But this can be very subtle and hard to track programming errors. These are not syntax errors but logical ones. So beware of using too many global variables. Now have a look on the other side of the picture. While writing functions, we pass values to them. So instead of passing the value of *i*

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

again and again, we declare it as global. Now it is available in the function, leaving no need of passing it.

Let's now come to the next variety of variables. The variables, defined in the main function are local to the function main. It means that they are accessible in all parts of the main function. Their values can be assigned, used in computations and later displayed. When we enter some function other than main, these variables are not accessible there. They are hidden. The global and the local variables, declared in a function are visible. The arguments passed to a function, are also visible. We pass the parameters through stack. Parameters are written on the stack. Later, the function is called which reads from the stack and makes a temporary copy for its use. The variables, declared and used inside the function are called automatic variables. They automatically come into being when the function is called. When the function finishes, these variables are destroyed. So automatic variables are created constantly and destroyed all the time. Here, we are talking about variables ordinary as well as user defined objects. Their behavior is same. They are automatic when the function is called, memory is allocated normally on the stack at the same time and used. When the function exits, these variables are destroyed. What happens if we want that when the function exits, some value, computed inside the function, is remembered by the function and not destroyed. This should not be visible by the other parts of the program.

Let's consider the example of a refrigerator. When we open the door of a refrigerator, the light turns on and we can see the things inside it. However, on closing the door, the light turns off. Do we know that light is off because whenever we open the door the light is on. When we close the door what is inside. We do not know. May be things magically disappear. When we open the door, magically, the things are at their position. You can think of this like a function. When we enter in the function, these automatic variables are available there and visible. When we came out of the function, it is like closing the door of the refrigerator and the light is turned off. We cannot see anything. Function goes one step ahead of this and it actually destroys all the variables. Whereas, in the refrigerator, we know that things are there. Somehow we want the function to behave like that. Outside the refrigerator, these things are not available. We can not access them. Let's say there is a bottle of water inside the refrigerator. You open the door and place it some other place. Next time, when you will open the door, the bottle is seen at the same position where you have moved it. It would not have moved to some other position. If you think of automatic variables, suppose we say inside the body of the function *int i = 0;* Every time the function is called and you go into the function where *i* is created. It has always the value 0 to start with and later on we can change this value.

What we want is that whenever we go back into the function, once we call the function like we open the door of the refrigerator and move the bottle to some other place and close the door. So we made one function call. Next time, when we call the function, the bottle is found in its new place. In other words, if we have defined an integer variable, its value will be set at 10 in the function when we return from the function. Next time when we call the function, the value of that integer variable should be 10 instead of 0. We want somehow to maintain the state of a variable. We want to maintain its previous history.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

If we declare a global variable, the state would have been maintained. The global variable exists all the time. Whatever value is set to it, it is there and accessible from any function. The drawback is that variable exists even when we don't want it. Static keyword allows us a mechanism from getting away of the downside of the global variables and yet maintaining a state inside a function. When we visit, it is found out what are its values before that we go ahead with this value. For this, whenever we declare a variable inside the function, *static* keyword is employed before the variable declaration. So we write as:

```
static int i;
```

That declares *i* to be a static integer inside the function. Think about it. Should we declare static variables inside the main function? What will happen? 'main' itself is a function so it is not illegal. There is no objective of doing this in main because main is a function from where our programs start and this function executes only for once. So its state is like an ordinary variable, declared inside main. It is only relevant for the called functions. We write inside the function as *static int i;* while initializing it once. It will be created only once irrespective of the number of function calls. Now once it is created, we increment or decrement its value. The function should remember this value. The programmer may go out of the function and come back into it. We should get the value that should be same as that at the time of leaving the function. It is necessary for the static variables that when these are created, they should be initialized. This initialization will be only for once for the complete life cycle of the program. They will be initialized only once.

Here, we have to take care of the subtle difference. In case of ordinary variable declaration, we should initialize them before using. If you have to initialize an *int* with zero, it can be written as *int i;* and on the next line *i = 0;* But in case of static variables, we have to use a different type of initialization. We have to use it as *static int i = 0;* It means that creation of *i* and the allocation of memory for it takes place simultaneously. It is initialized and the value 0 is written. This is initialization process. If somewhere in the function, we have statement *i = 10;* it will not be treated as initialization. Rather, it is an assignment statement. Here we want that as soon as the variable is created, it should be initialized. This initialization will be only for once for the lifetime of the program and it takes place when first time we enter in to the function. However we can manipulate this variable as many times as we want. We can increment or decrement it. However, it will remember its last value. How does this magic work? So far, we have been talking about the stack and free store. There is another part of memory, reserved for the variables like static variables. On the stack, automatic variables are being created and destroyed all the time. The heap or free store has the unused memory and whenever we need memory, we can take it from there and after use return it. This is the third part which is static memory area where static variables are created and then they exist for the rest of the program. These variables are destroyed on the completion of the program. So they are different from automatic variables which are normally created on stack. They are different from dynamic variables that are obtained from free store.

To prove this whole point let's write a simple program to fully understand the concept and to see how this works. Write a small function while stating that *static int i = 0;* Here, we are declaring *i* as a static integer and initializing it with zero. Then write

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

i++; print the value of *i* using *cout*. Now this function just increments the value of *i*. This *i* is a static integer variable inside the function. Now write a main function. Write a loop inside the main and call this function in the loop. Let's say the loop executes for ten times. You will notice that whenever you go inside the function, the value of *i* is printed. The value of *i* should be printed as 1.2.3…10. If you remove the word static from the declaration of *i*, you will notice that every time 1 is printed. Why 1? As *i* is now automatic variable, it is initialized with zero and we increment it and its value becomes 1. *cout* will print its value as 1. When we return from the function *i* is destroyed. Next time when function is called, *i* will be created again, initialized by zero, incremented by 1 and *cout* will print 1. By adding the static keyword, creation and initialization will happen once in the life time of our program. So *i* is created once and is initialized once with the value of zero. Therefore *i++* will be incrementing the existing value. At first, it will become 1. In this case, function will return from the loop in the main program**,** call this function again**.** Now its value is 1, incremented by 1 and now the value of *i* becomes 2 and printed by *cout*. Go back to main, call it again and so on, you will see it is incrementing the last value. You can prove that static works.

Here is the code of the program:

```
/*  This is a simple program. This shows the use of static variables inside a function.
*/

#include <iostream.h>

void staticVarFun();
void nonstaticVarFun();

void main(void)
{
   cout << "\nCalling the function which is using static variable \n";
   for(int i = 0; i < 10; i++)
      staticVarFun();

   cout << " \nCalling the function which is using automatic variable \n";
   for(int i = 0; i < 10; i++)
      nonstaticVarFun();
}
// function definiition using static variables
void staticVarFun()
{
   static int i = 0;
   i++;
   cout << "The value of i is:" << i << endl;
}

// function definiition using automatic variables
void nonstaticVarFun()
{
   int i = 0;
```

vubookshoppk.com

```
    i++;
    cout << "The value of i is:" << i << endl;
}
```

The output of the program:

```
Calling the function which is using static variables
The value of i is:1
The value of i is:2
The value of i is:3
The value of i is:4
The value of i is:5
The value of i is:6
The value of i is:7
The value of i is:8
The value of i is:9
The value of i is:10

Calling the function which is using automatic variables
The value of i is:1
The value of i is:1
The value of i is:1
The value of i is:1
The value of i is:1
The value of i is:1
The value of i is:1
The value of i is:1
The value of i is:1
The value of i is:1
```

vubookshoppk.com

Static Objects

Let us look at some more uses of this keyword. As mentioned earlier that the user defined data types are the classes and objects that we created. These are now variables as for as we are concerned. If these are variables, then we can declare them as static. Now we have to be careful when we think about it. When we declared static int, we said that it should be initialized there. We initialized it with zero. What is the initialization of objects? We have defined a class and instantiated an object of that class. So we can say something like vehicle A or truck B where vehicle and truck are the classes which we have defined. 'A' and 'B' are their objects, being created in some function or main. When are these objects initialized? You know that the initialization is done in constructors. So normally C++ provides a default constructor. Here we have to write our own constructors as initialization can happen only once, if declared static. Again we are talking about these static objects inside a function instead of the main. These objects should maintain their values while getting out of the function.

Whenever we create a static object, it must be initialized. Most of the time, we want that when the object of our class is created, its data members should be initialized by some value. For this purpose, we have to provide a constructor so that whenever an object is created, its data members are initialized. Only then it will work. Otherwise we will have problems. We may want to do as truck A, but our constructor takes some arguments. Now how this object will be created. How many wheels this truck will have? How many seats will be there? We have a solution to overcome this problem. Define a constructor which takes arguments and provides the default value to it simultaneously. If you provide a constructor with default values, then the object which is created will automatically get these values. If you write *truck A(4, 6)*, there may be some constructor which will initialize it with 4 wheels and 6 seats. But the point to remember is if you ever go to use a static object, it is necessary to provide a constructor with default arguments so that the object which you have created is initialized properly. Other than that the whole behavior of a static object is exactly the same as we have a static variable of an ordinary data type or native data type. Static variable means maintaining the state of a variable. It exists and lives around even when we are outside the function. It is an alternative to using a global which exists even when we don't want it. Now we try to learn about the destructors of static objects. If you create an object inside a function as *truck A*, when the function finishes, the object *A* will be destroyed. Destructor for this static object will be called.

To prove this write a class, inside the constructor. Also write a *cout* statement which should print 'inside the constructor of 'and the name of the object which will be passed as an argument. In the destructor write a cout statement as *cout <<" Inside the destructor of " << name*, where name will tell us that which object is this. Now experiment with it. Declare a global variable of this class before main as *truck A('A')*. When the constructor for this object is called the line 'Inside the constructor of A' will be displayed. Now within the main function, declare another object as ordinary variable i.e. *truck B('B')*. Its constructor will also be called. You will see it. Write a small function and create another object within that function as *truck C('C')*. Define another function and declare a static object in it as *truck D('D')*. Call these two functions from main. Now compile and execute this program, as we have written *cout* statements inside the constructor and destructor. Now you will be able to determine which object is being created and which one being destroyed. Here you will also notice that first of all global object *A* will be created. There is going to be a line 'Inside the constructor of object A'. After that, object *B* will be created, followed by the display of constructor *cout* line. From main, we are calling function *F* which is creating object *C*. So object *C* will be created then. What next? The function *F* will finish and the control go back to main. If the function *F* finishes, its local data will be destroyed. So the object *C* will be destroyed. Here, you see it on the screen 'Inside the destructor C'. After this the function *G* will be called and we will have 'Inside the constructor for D'. This object D is a static object. Now when the function *G* finishes, you will not see the destructor line for object *D*. After this, the main function finishes and the destructors will be called for objects *A* (which is global), object *B* (which is inside the main) and object *D* (which is created as static inside the function G). In which order these will be called?. If you look at this very simple program, you will find that the last object to be created was the static object inside the function *G*. Should that deleted first i.e. the destructor of object *D* should be called? Well actually not true, the local variables of main function will be first destroyed. Static objects remain for longer period of time. Later, the static object *D* will be destroyed and the

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

thing finally destroyed is the global object, which was created first of all. You will find that the destructor for object *A* is called. With this exercise, you will know the sequence in which things are created and destroyed. Another thing that you will notice is that when the function *G* finishes the static object is not destroyed.

The code of the program;

```
// An example of static objects, notice the sequence of their creation and destruction
#include <iostream>

// defining a sample class
class truck {
  private:
    char name; // Identifier
  public:
    // constructor displaying the output with the object name
    truck(char cc):name(cc) {
        cout << "inside the constructor of " << name << endl;
    }

    // distructor displaying the output with the object name
    ~truck() {
        cout << "Inside the destructor of " << name << endl;
    }
};
// defining a global object
truck A('A');

// a simple function creating an object
void f() {
  truck C('C');
}
// a simple function creating a static object
void g() {
  static truck D('D');
}

// main function
int main() {
  // an ordinary object
  truck B('B');
  // calling the functions
  f();
  g();
}
```

The output of the program:

```
inside the constructor of A
inside the constructor of B
inside the constructor of C
Inside the destructor of C
```

vubookshoppk.com

```
inside the constructor of D
Inside the destructor of B
Inside the destructor of D
Inside the destructor of A
```

Lets recap these concepts. When you declare a static variable (native data type or object) inside a function, it is created and initialized only once during the lifetime of the program and therefore it will be destroyed or taken out of memory only once during the lifetime of the program. So it is a good way of maintaining state. It is an alternative to using a global data type which has some side effects. In the main, we can write static variables but it is a meaningless exercise because these are exactly like ordinary variables inside main.

Static data member of a class

Lets talk about the keyword static inside the class. Static variables are used to maintain state. We are talking about the state in which we left the function. While extending the concept, we will go inside an object. Here, we should find certain things left exactly the way they were initially. So now we are talking of static data members inside a class. What does it mean?

Literally speaking, the word 'Static' means the stationary condition of things. Stationary for object or class? Here it will be stationary for the class. That means that static data will be created once and initialized once for that class. Therefore it is not related to the objects of that class. There is only one copy of the static data member inside a class. The copy is not repeated for the objects. Whenever we create an object of a class, the complete data structure is copied for that object and there is one copy of functions which the objects may use. Static members are single for the whole class in the static memory area. It will not be repeated whenever we create an object of the class.

Now the question arises when it will be created? When it will be initialized? And when it will be destroyed? Now these are on class level and not on object level. To understand this, we have to talk about the lifetime of the static data member. The lifetime of the static data member of a class is the lifetime of the program. In other words, when you include a class in the program as a class definition, the memory is allocated for its static data members. We have some techniques to initialize it. We initialize it only once. Initialization is done at file scope which means almost at the global scope. We initialize it outside of the main. The memory is allocated for these static members. No other copy can be created for them. Therefore we can create and initialize them outside of main. There is no object so far. How can we initialize its static data members?

Suppose we have a class truck as:

```
class truck{
   public:
                int wheels;
                int seats;
```

```
}
```

Now we refer the data members with the object as:

```
truck A;
A.wheels = 6;
A.seats = 4;
```

That's a way to refer to a data member. Here we are saying that we have some static data member of class and the object *A* has not been created yet. But we have the memory for the static members. Now we want to initialize that memory. How can we do that? We do this by using the scope resolution operator (::) and on its left hand side, we have class name and not the object name. On the right side, we write the name of the static data member. Suppose we have some static integer data member *i* in the class truck, so we can write it as:

```
truck::i = 10;
```

This initialization is taking place at file scope outside of the main .As it is happening only once in the program, it will not be executed again. It is being initialized once for the class. You can create as many object as you want. Objects can read and change that value.

Static data members of a class can be public or private. The objects of the class have access to them. They can manipulate it. But it is created and initialized only once. There is a single copy of these static data members regardless of how many objects of the class you create.

Let's take a look at the problems having static data members of a class. Suppose we have a class as 'savingsAccount'. We deposit money in that account. Some profit is also earmarked for it. Over a period of time, bank declares the rate of the profit. Profit rate is same for all PLS accounts. We have defined a class *savingsAccount* which have the information like person name, account number, current balance etc. We also have to keep the profit rate so that we can apply that on the account. Do we have different profit rate of every account? No, the bank has declared say 3% profit rate. That will be applied to all the PLS accounts. So we want that the profit rate should be defined at one place. It should be the part of the class but not defined for each object. So it is a good place to use a static variable as a data member of the class. We can initialize it at file scope as:

```
savingsAccount::profit_rate = 3.0;
```

We will write this before main function. As soon as, we compile the program and try to run it, the space is created in the static storage area. The above statement will initialize that static memory with 3.0. No savings account has been created yet. We will be creating saving accounts (object of class savingsAccount) in the main or some other function as account1, account2, etc. This profit rate will be available to every account. We can access it as:

```
account1.profit_rate;
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

and can use it in computations. This is legal but a bad usage. Why it is a bad usage? Suppose we write as;

```
account1.profit_rate = 4.0;
```

What will happen? Does the profit rate for only *account1* has been changed? No. There is only one copy of profit_rate for all the objects of this class. That means if an object manipulates the static data member, which it can through the member functions or directly depending on it is private or public. It is actually modifying the value of that static data member for all objects of this class. So don't assume that it will change profit_rate for one object. It is a legal but a bad usage. Always use it with the class name and not with the object. So you should access it as *savingsAccount::profit_rate*. It means you are resolving it at class scope. Be careful while applying it.

Let's consider another example. We have a class as *student* and a data member for how many students in the class. Now every time, a student enrolls in the course, we want that number of students should be incremented. Whenever a student withdraws, fails or passes out from the course, the number of students should be decremented. We want this to be inside the *student* class. How does it work? We define a static data member as *static int how_many;* and initialize it to zero as:

```
student::how_many = 0;
```

In the constructor of the class we write as:

```
how_many++;
```

This way, whenever a student object is created, *how_many* will be incremented. Whenever a student leaves the course, its destructor should be called. We will write in the destructor as:

```
how_many--;
```

So it's a good way of keeping track of how many objects of a particular type exist at this time inside the program. To display that we can write a member function that will display 'how_many'. The merit of this technique is that we have done all this work inside the class. We did not use two classes or global variable or go through the source code to count the number of students. Using these, you can make your program more and more dynamic. The usage of static is very import. So you should understand it clearly. Static is maintaining the state. The state may be how many students are in the class.

Today we have covered the parameter-less manipulators which will return the *ostream* object and *ostream* object is passed as an argument to them. Then we discussed about the static data, both at the ordinary level and then the static data members inside the class. These are very useful. As you write bigger and more complex programs, you will find that these concepts are very useful. Again from a generic prospective, you will be working hopefully in your professional career with many different languages. You have to understand that every language might

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

represent the static concept in a different way. But just knowing that concept empowers you and help you to understand more complex programming languages as well.

Lecture No. 39

Reading Material

Deitel & Deitel - C++ How to Program Chapter. 5, 6, 7

Revise old topics

vubookshoppk.com

Summary

51) Pointers
52) References
53) Call by Value
54) Call by Reference
55) Dynamic Memory Allocation
56) Assignment and Initialization
57) Copy constructor
58) Example
59) Rules for Using Dynamic Memory Allocation
60) Usage of Copy Constructor
61) Summary
62) Exercise

In this lecture, we will review the concepts like pointers, references and memory allocation, discussed in the previous lectures. The review of these topics will help enhance our understanding of the programming. Let's discuss these topics one by one.

Pointers

Pointer is a special type of variable that contains a memory address. It is not a variable that contains a value, rather an address of the memory that is contained inside a pointer variable.

In C++ language, variables can be without type. Either we can have a void pointer or these can be typed. So we can have a pointer to an integer, a pointer to a character and a pointer to a float etc. Now we have a user-defined data type, which we call classes, so we can have pointers to classes.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

While talking about pointers, we actually refer to pointing to an area in memory. A pointer to an integer, points to a location opted by an integer in memory. If there is an array of integers in the memory, we can still use a pointer to point to the beginning of the array. This is a simple manipulation. To have the address of a memory location, we use & sign. The '&' sign is also used in references. So we have to be very cautious while making its use. To further overcome this ambiguity, we will now recapture the concept of reference.

References

A reference can be considered as a special type of pointer as it also contains memory address. There are some differences between pointers and references. Pointers may point to nothing while references always have to point to something. A reference is like an alias for an object or a variable. The references should be used when we are implementing the call by reference. This helps us make our syntax easier as we can implement the call by reference with out using the * operator.

Call by Value

Whenever we call a function and pass an argument, an object or variable to the function, then by the default rule of C and C++, it is a call by value. It means that the original data remains at its place and a temporary copy of it is made and passed to the function. Whatever the function does with this copy, the original value, in the calling function, remains intact. This is a call by value.

Call by Reference

If we want a function to change something in the original object variable or whatever, that variable or object by reference would be passed. To do this, we don't make temporary copy of that object or variable. Rather, the address of the variable is sent. When the function manipulates it, the original object will be manipulated, effecting change in its values. The use of call by reference is also important for the sake of efficiency. If we have a large object, sending of its copy will be something insufficient. It will occupy a large space on the stack. Here, we can use call by reference instead of call by value only for efficiency while we need not to change the original object. For this, we use a keyword *const* that means that a *const* (constant) reference is being passed. The function can use its values but cannot change it.
Now we come to the dynamic memory allocation.

Dynamic Memory Allocation

In C language, we have a method to allocate dynamic memory. In it, while executing the program, we allocate some memory from the free store (heap) according to our need, use it and after using, send it back to the free store. This, dynamic memory allocation, is a very common function in programming. While writing C++ language, it was decided that it should not be implemented by a function call. There should be native operators, supposed to be very efficient. These operators are *new* and *delete*. We allocate memory with the *new* operator from the free store. It returns a pointer. For example, if we say *p* is a pointer to an integer, the statement will be written as

```
int *p ;
```

Now we write

```
p = new int ;
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

This statement performs the task in the way that it gets memory for an integer from the free store. There is no variable name for that memory location but the address of this location is stored in the pointer *p*. Now by using the syntax ***p*** (which means whatever p points to), we can manipulate that integer value. We can also write to and read from that location. This memory allocation is done during the execution of the program. Whenever we allocate memory with the *new* operator, it is our responsibility to de-allocate this memory after the termination of the program. To do this de-allocation, we have an operator *delete*. To de-allocate the memory, allocated with *p = new int ;* we will write

delete (p) ;

It will not delete the *p* rather, it will send the memory gotten and pointed by *p* back to the free store.
Same thing happens when we try to allocate more than a simple variable i.e. when we are trying to allocate arrays. When we use *new* operator to allocate a space for an array, we tell the size of the array in square brackets (i.e. []). We can write it like

p = new int [10] ;

This statement says *p* is pointing to an area of memory having the capability to store 10 integers. So there is an array of 10 integers whereas *p* is pointing to the beginning point of the array. Now we can access the elements of the array by manipulating the pointer *p*. To make the memory allocated for an array free, the syntax is a little bit different. Whenever, we allocate an array then to free it we write

delete [] p ;

This will free the array that is pointed by *p*. In this case, the space of 10 integers that was pointed by *p,* will be de-allocated despite the fact that we write empty brackets with the *delete* operator. This is due to the fact that C++ internally has the record that how much memory was allocated while allocating an array. After *delete,* the pointer points to nothing. So it's a free pointer and can be reused.
Now let's go on and look at a special type of objects. These are the objects with the data members as the pointers. In the previous lectures, we had discussed the example of a class Matrix. In that class, we said that it will be a two dimensional matrix while talking about its size etc. In this example, it will be a generic class besides being a matrix of 3 x 3. Now we want to see the Matrix class defined such a way that the programmer can take an object of it of any size at any time, say a matrix of 3 x 3, 5 x 5 or 10 x 10. It means that the size of the matrix should be variable. Now we have to bring every thing together in terms of how we declare and manipulate arrays inside a C++ program.
Whenever we declare an array, we have to mention its size. It is necessary, as otherwise no memory will be allocated, it's about the static memory allocation. In it, we declare like that it will be a two dimensional array of *m* rows and *n* columns. The compiler will allocate a memory for this array at the start of the program. Now we are talking about that at the compilation time, we don't know the size of the array. We want to allocate an array of the required size at run time. When the constructor of the class is called, the memory required by that object should be allocated at that time.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

For example, in case of Matrix class, we will like to tell the number of rows and columns of the object of the Matrix in the constructor, so that the constructor could allocate the memory for that object. A keyword *new* is used for this purpose. Here, we realize that in case of data member of this class Matrix, fixed rows and columns cannot be used in the class definition. We cannot define a two-dimensional array inside the class as it negates the concept of extensibility. Here, inside the class we define something like *int *m;* which means *m* is a pointer to an integer. It is the data member of the class. In the constructor of the class, we say that it will take two integer arguments, rows and columns. Whenever we declare an object, which is an instantiation of the class, the constructor of the class is called. Now in the constructor of this class we want to allocate memory from the free store equal to the memory required by *m* x *n* (m multiply by n) integers. So if we say a Matrix of 3 rows and 3 columns then we require memory for 9 (3 * 3) integers. If we say four rows and five columns then we require a memory for 20 (4 * 5) integers. Now it is very simple, inside the constructor we will write

```
m = new int [rows * columns] :
```

Thus we created a Matrix to which we tell the number of rows and columns, through variables, during the execution of the program. When we created an object of the class its constructor is called to which the variable values are passed and by multiplying these variables (number of rows and columns), we allocate a memory for these integers by the *new* operator and its address is assigned to *m*. Thus the object is initialized and the required memory is available to it. Now we can use it as a two dimensional array and can put values inside it. Now the class definition of the class Matrix can be written as under.

```
class Matrix
{
        private:
                int *m;
                int row, col;
        public:
        Matrix(int rows, int cols)
        {
                        m = new int[rows * cols];
        }
};
```

There is a requirement that if the constructor of a class allocates the memory, it is necessary to write a destructor of that class. We have to provide a destructor for that class, so that when that object ceases to exist, the memory allocated by the constructor, is returned to the free store. It is critically important. Otherwise, when the object is destroyed, there will be an unreferenced block of memory. It cannot be used by our program or by any other program. It's a memory leak that should be avoided. So whenever we have a class in which the constructor allocates dynamic memory, it is necessary to provide a destructor that frees the memory. Freeing the memory is an easy process. We have no need to remember that how many rows and columns were used to allocate the memory. We simply use the *delete* operator with empty brackets and the pointer that points to the allocated memory. We write this as follows

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
delete [] m ;
```

This statement frees the allocated memory (whatever its size is) that is being pointed by *m*.

Assignment and Initialization

Let us discuss the assignment and initialization. We do initialization as

```
int i = 0 ;
```

This is declaring an integer and initializing it. The second way to do this is

```
int i ;
i = 0 ;
```

This has the same effect as the first statement but the behavior is different. At first, a space is allocated for *i* before assigning a value to it..
The same applies whenever we create an object. We can either create an object, initialize it at the creation time that means constructor is being called, or we can create an object and then assigns values to its data members later. This is usually done either by set functions or with the assignment statements. Here the thing to differentiate is that if we have two objects of a class, say Matrix, *m1* and *m2*. We have, in some way, created *m1*, its rows and columns have been allocated, and values have been put in these rows and columns. Now somewhere in the program, if we write

```
m2 = m1 ;
```

It is an assignment statement. If we have not defined the overloaded operator for assignment, the default assignment of C will be carried out. The default assignment is a member-to-member assignment. Now let's again look at the construct for the Matrix class. In it, we have only one data member i.e. a pointer to an integer. We have written *int *m ;* in the class definition. So in *m1*, there is a pointer to an integer but *m1* is a fully qualified developed object. It is, let's say, a 5 x 5 matrix and has values for its entities. Now when we write

```
m2 = m1 ;
```

m2 has also a pointer *m* to an integer as its own data member. If we have not written an overloaded assignment operator for this class, the value of *m* of the object *m1* will be assigned to *m* of the object *m2*. Now we have two objects, having pointer variables that hold the same address. So these are pointing to the same region in the memory. There arise many problems with this. Firstly, if we destroy *m1*, the destructor of *m1* is called and it frees the memory. So the memory being pointed by *m* of the object *m1* has gone back to the free store. Now what happens to *m2*? There is a pointer in *m2*, pointing to the same memory, which has been sent to the free store by the destructor of *m1*. The memory is no longer allocated. It has been sent to the free store. So we have a serious problem. We don't want two objects to point to the same region in the memory. Therefore, we write an assignment operator of our own. This assignment operator is such that whenever we do some object assignment, it allocates separate

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

memory for one object and copies the values of the second object into that space. Thus the second object becomes an independent object. So we have to be careful. Whenever we have a class with a dynamic memory allocation, there is need for writing an assignment operator for it.

Copy Constructor

Now we will see what a copy constructor is? We have discussed the dangers that a programmer may face during the process of assigning the objects. The same danger comes to the scene, when we pass an object like the Matrix class to a function that does some manipulations with it. Suppose, we have a function that takes an object of a class Matrix as an argument. The default mechanism of calling a function in C or C++ is call by value. Now what is the value for this object, being passed to the function? The values of the data members of the object will be placed on the stack. The function will get a temporary object, as it is a call by value. The original object remains intact. The values of data members of that temporary object will be the same as the values in the original object. Now if it is a simple class, there will be no problem. However, if there is a class with a pointer as its data member and that pointer has allocated some memory, then the value of that pointer is copied. This value is passed to the function and not the memory. Now in the function, we have a temporary object that has a pointer as data member, pointing to the same memory location as the original object. If we are just reading or displaying that object, there is no problem with it. If we change the values of the object, the values in the temporary object get changed. However, when we manipulate the pointer, it changes the values in the memory of the original object. This change in the original values is the mechanism of the call by reference and here we have done call by value. We know that a temporary object is passed to the function. So how we get around this problem? The way to resolve this problem is to create a complete copy of the object. We want that in this copy of the object the pointer value must not point to the same memory location. Rather, it must point to the memory location of its own. So we want to have a copy constructor. That means a constructor that will create a new object with a full copy of the other object. This is also known as deep copy as opposed to shallow copy. The shallow copy makes a member-to-member copy of the object without taking care whether the pointer is a pointer or an ordinary variable. The copy of the ordinary variables is perfectly valid and legal. But if we make a member copy of a pointer, there may be the problem in which a new pointer variable is created with the same value, without creating a new area of memory. Therefore, we have to write something special that is called copy constructor.

The basic line in the syntax of the copy constructor is that we are trying to create a new object by passing an object of the same class as the argument. So we should think of a constructor. For example if we have the class Matrix, its prototype will be as under.

```
Matrix (Matrix &) ;
```

The '&' sign shows that a reference of the object of type Matrix will be passed. Now whenever we write a copy constructor, there is need to be very cautious. We have to allocate a new memory for that copy. When we go into this copy constructor, at first, it is determined that how much memory the original object has allocated? Since we pass the object, so all the queries might have answers. For example, in case of Matrix class, we can find the number of rows and columns. We create a temporary object

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

inside the constructor and allocate it a memory. Then the values of memory of the original object are copied into this memory. Now the pointer of this new object will point to this new location. As the constructor returns nothing and just creates a new object, so there is no return statement in the constructor. Thus, this copy constructor is completed. The syntax of this constructor is given below

```
Matrix::Matrix ( const Matrix  &other )
{
        size = other.size ;                 // size is a function
to determine the
    memory allocated by object
        m = new int [size] ;
        copyvalues ( m, other ) ;
}
```

In this case, it creates a new object that actually creates memory. It does not make the shallow copy so there is no copy of the pointer value. In fact, the pointer has a new value. But the values pointed to by this pointer (i.e. the values in the new allocated memory) are the copy of the values in the memory of the original object, and then this fully constructed object is returned.
Now we have the facility of copy constructor. With it, we can define new objects based on the existing objects. This copy constructor is necessary for the objects in which the memory is allocated dynamically. We can use this copy constructor without causing any problems. Suppose we have written as

```
Matrix a (3,3) ;
```

We have defined a constructor that takes two arguments rows and columns, as a matrix 'initializer'. We allocate memory for these rows and columns and create an object. Then somewhere, later in the program, We write

```
Matrix b (a) ;
```

This statement makes a complete copy of already created object *a* and a new object *b* is created. This new object has its own pointer and own memory allocation. This memory location is different from the memory allocated inside the matrix *a*. Now if *a* dies then *b* is still a valid object. So a copy constructor is critically useful. It is used when we want to create a duplicate copy of an object. It is always used whenever we want to do a call by value into a function to which an object is being passed, and the object is of a class in which we have allocated the memory dynamically.

Example

Let's look at an example to further understand these concepts. We write a class *String*. It has a data member *c* that is a pointer to a character. We write a member function (copy function) of the class, which can copy values in the character array of the class. There is a constructor that will allocate a space for the character arrays i.e. string. The starting address of this array will be stored in data member of the class i.e. in a pointer. We have not written any copy constructor. Now we want to make two objects of this *String* class, say, *s1* and *s2*. We create object *s1* and assign a string to it. We write it as

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

String s1 (“test1”) ;

Now after this we write

String s2 = s1 ;

Thus we create a new object *s2*. The values of *s2* are initialized with the values of *s1*. As we have written no copy constructor, C will provide the default copy constructor. Now if we display the string of *s2*, it will be the same as of *s1*. Now use the copy function to assign new values to the string inside the object *s1*. So we write

s1.copy(“A new string”) ;

Thus we write a new string in *s1*. Now again if we display the string of *s2* by writing

s2.print ;

We will see that it displays the same value, assigned to *s1* in the previous statement. The reason is that the default copy constructor has done member-to-member copy. It has copied the value of the character pointer to the pointer of *s2* and thus both pointers are pointing to the same memory location.
Now there is need of providing a copy constructor for a class like this. We also have to provide a destructor as we are doing memory allocation inside the constructor of the class.

Following is the code of the example in which we provide a copy constructor. We create an object based on an existing object. The copy constructor creates an object with full copy of the existing object with its values in a new memory location.

```
/*This program has a copy constructor and demonstrate the use of it.
We create a new object by passing it an existing object, this calls
the copy constructor and thus creates a complete copy of the passing
object, and has its values in new location of memory.
*/

#include <iostream.h>
#include <stdlib.h>

// class definition
class String
{
        char *c;
    public:
         // copy function
         void copy (char *s)
        {
             c = s ;
         }
         // getting the length of the string
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
        int length ()const
        {
            return strlen(c);
         }
         //constructors
        String ();
        String (const char *s)
        {
              c = new char [ 30 ];
               strcpy (c, s);
         }
        // copy constructor
        String( const String &other );

        //display the string
        void print()
        {
            cout << c << endl ;
         }
         //destructor
         ~String()
         {
             delete []c ;
          }
};
// definition of copy constructor
String::String( const String &other )
{
        int length;
        length = other.length();
        c = new char[length + 1];
        strcpy( c, other.c );

}

main ()
{
        String s1("test1");
        cout << "The string of s1 is    "  ;
        s1.print();
        String s2(s1);
        cout << "The string of s2 is    " ;
        s2.print();
        s1.copy("A new string"); // assign new value to string s1
        cout << "The string of s1 is     " ;
        s1.print();
        cout << "The string of s2 is    " ;
        s2.print();             //s2 has its own previous value
}
```

vubookshoppk.com

The following is the output of the program which shows the use of copy constructor.

```
The string of s1 is    test1
The string of s2 is    test1
The string of s1 is     A new string
The string of s2 is    test1
```

The other affected part is the assignment operator itself. We know that there are dangers in the assignment operators of a class in which memory is being allocated. We cannot write an assignment operator for such a class blindly. When we write an assignment operator for such a class, that operator must first look at the fact whether there is self-assignment being done.

Suppose we have an integer *i*. We have written as

```
int i ;
i = 10 ;
```

And down to this we write

```
i = i ;
```

There is no problem with it. It is a legal statement. It is complicated if we do such assignment through pointers. In such a case, pointer is pointing to itself and even it has no problem. But when we do this with objects that have allocated dynamic memory, the method of assignment is changed. Let's take the example of Matrix. We have an object of Matrix, say *m1*, which has three rows and three columns. Another object, say *m2*, has five rows and five columns. Somewhere in the program we write

```
m1 = m2 ;
```

Here *m2* is a big object while m1 is a small one. We want to assign the big object to the smaller one. The assignment operator for this type of class, first de-allocates the memory reserved by the left-hand side object. It frees this by the *delete* operator. Then it will determine the memory required by the object on right hand side. It will get that memory from the free store by using *new* operator. When it gets that memory, it will copy the values and thus the statement *m1 = m2*; becomes effective. So assignment has a requirement.

Now if we say

```
m1 = m1 ;
```

We have defined assignment operator. This operator will delete the memory allocated by *m1* (i.e. object on L.H.S.). Now it wants to determine the memory allocated by the object on the right hand side, which in this case, is the same i.e. *m1*. Its memory has been deleted. So here we get a problem. To avoid such problem, whenever we write an assignment operator, for objects of the class that has done memory allocation. After this, we do other things.

We have discussed the example in which we create an object of Matrix. We create it using a copy constructor by giving it another object. The syntax of it we have written as

```
Matrix m2 (m1) ;
```

This is the syntax of creating an object based on an existing object. We can write it in the following fashion.

```
Matrix m2 = m1 ;
```

While this statement, we should be very clear that it is not an assignment only. It is also a construction. So whenever we are using initialization, the assignment operator

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

seems as equal to operator. But actually assignment operator is not called. Think about it logically that why assignment operator is not called? The assignment operator is called for the existing objects. There should be some object on the left-hand side, which will call the assignment operator. When we have written the declaration line

Matrix m2 = m1 ;

The m2 object has not constructed yet. This object of Matrix does not exist at the time of writing this statement. So it cannot be calling the assignment function or assignment operator. This is an example of the use of a copy constructor. Thus, there are two different ways to write it. Remember that whenever we create an object and initialize it in the declaration line, it calls the copy constructor.

Let's talk about another danger faced by the programmers when they do not provide copy constructor. The ordinary constructor is there which allocates memory for the objects of this class. Suppose we do a call by value to a function. Here, we know that a temporary copy of the object is made and provided to the function. The function does manipulations with this copy. When the function returns that temporary copy is destroyed. As no copy constructor is there, a shallow copy, with values of pointers, is made. The destructor should be there as we do memory allocation in the class. Now suppose that there is a destructor for that class. Now when this temporary object destroys its destructor executes and de-allocates the memory. Now as it was a shallow copy so its pointers were pointing to the same memory as of the original object. In this way, actually, the memory of the original object is de-allocated. So the pointer of the original object now points to nothing. Thus, in the process of function call, we destroyed the original object as it is an invalid object now. Its pointer is pointing to an unknown memory location. This is a subtle but very critical. This can be avoided by providing a copy constructor, which actually constructs a fully formed object with its own memory. That temporary object will go to the function . When it is destroyed, its destructor will de-allocate this memory. However, the original object will remain the same.

Rules for Using Dynamic Memory Allocation

Whenever we have a class in which we do dynamic memory allocation, there are some rules that should be followed.

First, we must define a constructor for it. Otherwise, we will not be able to carry out dynamic memory allocation. This constructor should be such that it gets memory from the free store, initializes the object properly, sets the value of the pointer and returns a fully constructed object.

Secondly, we must write an assignment operator for that class. This assignment operator should first check the self-assignment and then make a deep copy.. So that a properly constructed object should be achieved..

Thirdly, as we are doing dynamic memory allocation in the constructor, it is necessary to provide a destructor. This destructor should free the allocated memory.
These three rules are must to follow.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Usage of Copy Constructor

Let us see where the copy constructors are being used?
First, it is used explicitly at some places, where we write

```
Matrix m2 (m1) ;
```

This is an explicit call to a copy constructor. Here *m1* is being passed by reference. If we want that there should be no change in *m1*, then it is necessary to use the key word *const* with it to prevent any change in *m1*. The presence of this key word means that the constructor will do nothing with the object, being passed to it. The use of copy constructor in this explicit way is clear.

The second way to use the copy constructor is by writing the declaration line as

```
Matrix m2 = m1 ;
```

Here the use of copy constructor is not clear. It is not clear by the statement that copy constructor is being used. It seems an assignment operator is being used. Be careful that it is not an assignment operator. It is a copy constructor.

The third use of the copy constructor is calling a function and passing it an object by value. If we have provided a copy constructor, it will be called automatically and a complete temporary copy (with memory allocation) of the object is given to the function. If we do not provide copy constructor, the call by value functions will create problems. In the function, if we change any value of the object, it will change the value in the original object.
In function calling, when we do the call by value, the copy constructor is called. On the other hand, in call by reference, copy constructor is not called and the address of the original object is passed.

Summary

A pointer is a variable that holds memory address. The & operator is used to get the address of a variable or an object. The & sign is also used as a short hand for a reference. Whenever we have a & sign in the declaration, it implies a reference. Whenever we have & sign on right hand side of an assignment operator, it is taken as address of an object. We can do dynamic memory allocation by using pointers.
In C++ language, we have two very efficient operators provided which are *new* and *delete*. We use the new operator for obtaining memory from the free store. It returns a pointer to the allocated memory. We store the value of this pointer in a pointer variable.
In a class, which allocates memory dynamically, there is a data member i.e. a pointer. When we create an object of the class at run time, it will allocate memory according to our requirement. So there is no waste of memory and the situations in which we want to store large data in small memory or vice versa are prevented. So we do dynamic memory allocation inside these classes.
Whenever we have dynamic memory allocation inside a class, we have to provide few things. We must provide a constructor that does the memory allocation for us producing a well-formed object.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

We must provide a copy constructor that is able to create fully formed copies of the objects. That means is should not only make the copies of the values of the pointers but it should give the pointers new values by allocating new memory for the object. And should copy the values of the original object into this new memory location.
We must provide an assignment operator. This operator should be able to check the self-assignment and can assign one object to the other in a proper fashion using the concept of deep copy and not a shallow copy. So we allocate memory space then copy element by element in this allocated memory.
And finally we must do de-allocation which means whenever we destroy an object and it goes out of scope, we should free the memory allocated by that object. To do the memory de-allocation we must provide the destructor of the class in which we free (delete) the memory by using deletes operator.

Exercise

You should write small programs to examine the working of these rules. You can check this if we allocate memory and do not delete it in the destructor. Then the next time, when we execute the program it will allocate a new memory. We can find that which memory is assigned by displaying the value of the pointer (not the value it points too). It will be a number with 0x-notation i.e. it will be in hexadecimal. We don't care about the exact value but we will find that if we have provided a proper destructor. Then on the same computer, in the same session, we execute the program, a specific address of memory will be assigned to the program. With the proper destructor, we stop the program and then again start it. Nine out of ten times, we get the same memory. That means we will see the same address. Nine times out of ten is because the operating system can use this memory somewhere else between the times of two executions of the program. If we do not provide a destructor i.e. we do not deallocate the memory, it is necessary that each time we will get a new memory. The previous memory is being wasted. You can prove it by yourselves by writing small programs.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Lecture No. 40

Reading Material

Deitel & Deitel - C++ How to Program ***Chapter 7***
7.3, 7.4

Summary

- Objects as Class Members
- Example 1
- Example 2
- Advantages of Objects as Class Members
- Structures as Class Members
- Classes inside Classes
- Tips

vubookshoppk.com

Objects as Class Members

A class is a user defined data type and it can be used inside other classes in the same way as native data types are used. Thus we can create classes that contain objects of other classes as data members.
When one class contains objects of other classes, it becomes mandatory to understand how and in what sequence the contained and containing objects are constructed. An important point in construction of an object is that the contained data members of the object (regardless whether they are native or user defined data types) are constructed before the object itself. The order of destruction of an object is reverse to this construction order, where the containing object is destroyed first before the contained objects.
To elaborate the construction and destruction orders of objects, we take a class *A* and contain its instance (object) in another class *B*.

```
/* This program illustrates the construction and destruction orders of objects. */

#include <iostream.h>

class A
{
```

```
   public:
   A()
   {
      cout   <<   "\n A Constructor ...";
   }

   ~A()
   {
      cout   <<   "\n A Destructor ...";
   }
};

class B
{
   public:
         B()
         {
            cout   <<   "\n B Constructor ...";
         }

         ~B()
         {
            cout   <<   "\n B Destructor ...";
         }

   private:
         A   a;
};

void main(void)
{
   B   b;
}
```

The output of this code is as follows:

```
A Constructor ...
B Constructor ...
B Destructor ...
A Destructor ...
```

In the code above, we have contained an instance of the class *A* inside class *B*. In the *main* function, we have only created an object of the class *B*.

From the output, we can see the first line that the contained object *a*'s default constructor is called before the default constructor of the class *B*. At destruction time, the destructor of the class *B* is called first before *A*'s. Note that the contained object '*a*' of class *A* is constructed by calling the default constructor. Hence, we have found one way of constructing contained objects by means of default constructors and then setting the values of data members by calling setter methods of the object. But this is cumbersome and wasteful, we have a better way provided by the language to initialize

vubookshoppk.com

contained objects at construction time using the *initializer list*. Initializer list is used to initialize the contained objects at the construction time.

Example 1

Let's take a class of *PersonInfo* that stores *name*, *address* and *birthday* of a person. This *PersonInfo* class contains an instance of our veteran *Date* class to store *birthday* of a person.

```
class   PersonInfo
{
    public:
        // public member functions...
    private:
      char   name[30];
     char   address[60];
     Date   birthday;          // member object
};
```

This declaration specifies a *Date* object *birthday* as a *private* data member. Note that no arguments are specified in the declaration of *birthday*. However, this does not mean that the default constructor is called when the *PersonInfo* object is constructed but we can always specify a member initializer to call a parameterized constructor.
A colon is placed after the parameter list of the containing class's constructor, followed by the name of the member and a list of arguments as shown below:

```
class   PersonInfo
{
      public:
         PersonInfo( char * nm, char * addr, int   month, int   day, int  year );
         // ...
      private:
         // ...
};

PersonInfo::PersonInfo( char * nm, char * addr, int month, int day, int year )
   : birthday( month, day, year ) // Member initializer
{
   strncpy( name, nm, 30 );
   strncpy( address, addr, 60 );
}
```

Note that there are five parameters inside *PersonInfo* constructor including the three parameters *month*, *day* and *year* parameters to be passed to *Date* class's parameterized constructor as *birthday(month, day, year)*. We are using the initializer list, therefore, there is no need to call setter methods of the *Date* class to initialize the *birthday* object. Similarly, multiple contained objects can be initialized by using comma separated initializers. The order of the execution of initializers is the same as the order of declarations of objects inside the outer class. To confirm about the order

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

of execution, let us have another *Date* object *drvLicenseDate* declared after *birthday* object in the *PersonInfo* class:

```
 /* This program illustrates the initializer list, order of execution of constructor's inside the list. */

#include   <iostream.h>
#include   <string.h>

class   Date
{
        public:
          Date( );
          Date(int   month, int   day, int   year);
          ~Date ( );

        private:
                int   month, day, year;
};

Date::Date( )
{
   cout << "\n Date -- Default constructor called ...";
   month   =   day   =   year   =   0;
 }

Date::Date(int  month, int   day, int   year)
 {
   cout   <<   "\n Date -- Constructor with month="   <<   month
         << ", day= "   <<   day   <<   ", year= "   <<   year   <<   " called ...";
   this->month   =   month;
   this->day   =   day;
   this->year   =   year;
  }

Date::~Date ( )
 {
        cout   <<   "\n Date -- Destructor called ...";
 }

class PersonInfo
{
public:
   // public member functions...
   PersonInfo( char  * nm, char  * addr, int  month, int  day, int  year,
                    int  licMonth, int  licDay, int  licYear );
   PersonInfo::~PersonInfo();

private:
   char   name[30];
```

vubookshoppk.com

```
    char   address[60];
    // member objects
    Date   birthday;
    Date   drvLicenseDate;
};

PersonInfo::PersonInfo( char  * nm, char  * addr,  int  month, int  day, int  year,
                        int  licMonth, int  licDay, int  licYear )
    : drvLicenseDate( licMonth,  licDay,  licYear), birthday( month,  day,  year )
          // Above line is initializer list
{
    cout   <<   "\n PersonInfo -- Constructor called ...";
    strncpy( name,  nm,  30 );
    strncpy( address,  addr,  60 );
}

PersonInfo::~PersonInfo()
{
    cout   <<   "\n PersonInfo -- Destructor called ...";
}

main(void)
{
    PersonInfo   pi("Abbas", "12-Y, DHS, Lahore, Pakistan", 12, 12, 1972, 12, 10, 1992);
}
```

The output of this program is:

```
Date -- Constructor with month=12, day= 12, year= 1972 called ...
Date -- Constructor with month=12, day= 10, year= 1992 called ...
PersonInfo -- Constructor called ...
PersonInfo -- Destructor called ...
Date -- Destructor called ...
Date -- Destructor called ...
```

Because *birthday* is declared before *drvLicenseDate*, it is clear from the output that the constructor for *birthday* is called first and then for the *drvLicenseDate* object, although *drvLicenseDate* is present before *birthday* in the initializer list.

Example 2

Let's take another example to work with the size of a matrix. We declare a *Column* class first then a *Row* class. *Row* class contains an instance of *Column* class to store the number of columns (number of elements) inside one *Row* instance. Further, the *Matrix* class contains an instance of *Row* class. See the code below.

```
/* Program to illustrate the initialization lists, construction and destruction sequences of
contained and containing objects. */
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
#include   <iostream.h>
#include   <stdlib.h>

class   Column
{
   private :
      int   size ;

   public :
      Column ( int   size )
      {
            cout   <<   "Column created"   <<  endl   <<   endl ;
            this->size   =   size ;
      }
      ~Column ( )
      {
            cout   <<   "Column destroyed "  <<   endl   <<   endl ;
      }

      void   showSize ( ) ;
      void   setSize ( int ) ;
};

void Column :: showSize ( )
{
   cout   <<   "Column size is : "   <<   size   <<  endl   <<   endl ;
}
void Column :: setSize ( int   sz )
{
   size   =   sz ;
}

class Row
{
   private :
      int   size ;
      Column   col ;
   public :
      Row ( int   rowSize, int   colSize ) : col( colSize )
      {
            cout   <<  "Row created"   <<   endl   <<   endl ;
            this->size   =   rowSize ;
      }
      ~Row ( )
      {
            cout   <<   "Row destroyed "   <<   endl   <<   endl ;
      }
      void   showSize ( ) ;
      void   setSize ( int ) ;
};
```

vubookshoppk.com

```
void   Row :: showSize ( )
{
    col.showSize ( ) ;
    cout   <<   "Row size is : "   <<   size   <<   endl   <<   endl ;

}
void Row :: setSize ( int sz )
{
   size   =   sz ;
}

class   Matrix
{
   private :
      Row   row ;
   public :
      Matrix ( int   rowSize, int   colSize ) : row( rowSize,  colSize )
      {
            cout   <<   "Matrix created"   <<   endl   <<   endl ;
      }
      ~Matrix ( )
      {
            cout   <<   "Matrix destroyed"   <<   endl   <<   endl ;
      }
      void   displayMatrixSize ( ) ;

} ;

void Matrix :: displayMatrixSize ( )
{
   row.showSize ( ) ;
}

void   f( )
{
   Matrix   matrix(3, 4) ;
   matrix.displayMatrixSize ( ) ;
}

int   main()
{
   f( );
   system("PAUSE");
   return   0;
}
```

The output of the program is as follows:

```
Column created
```

vubookshoppk.com

```
Row created

Matrix created

Column size is : 4

Row size is : 3

Matrix destroyed

Row destroyed

Column destroyed

Press any key to continue . . .
```

Notice the construction sequence of objects. In order to create a *Matrix* object, a *Row* object is created first and to create a *Row* object, a *Column* object is created. So the contained object *Column* is constructed first of all, then comes the *Row* object and finally the *Matrix* object. At destruction time, the very first object to destroy is the last object constructed, which is the *Matrix* object. The second object destroyed is *Row* object and then the *Column* object at the end. See also the use of initializer list in the code, how the *colSize* and *rowSize* arguments are passed to the constructors.

The *public* data members of a contained object can also be accessed from outside of the containing class. For example, if *row* object inside *Matrix* class is declared as *public* and has a *public* variable named *size* then it can be accessed using the dot operator (".") as:

```
int   main ( void )
{
        Matrix   matrix ( 4,  5 ) ;
        Matrix.row.size   =   8 ;
}
```

Advantages of Objects as Class Members

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

It is a way of reusing the code when we contain objects of our already written classes into a new class. For example,* Date *class can be used as data member of* Student, Employee *or* PersonInfo *class. In this approach, we don't have to test our previously written classes again and again. We write a class, test it once and add it into our components library to use it later.

It gives clarity and better management to the source code of our programs when we break up problems into smaller components. The smaller components can be managed independently from their contained objects forming their own classes. For example, in the previous example program, *Matrix* was subdivided into *Row* and *Column* classes.

When we declare an object as a constant data member inside a class then that constant object is initialized using the initializer list. Therefore, a class, whose object is contained as *const* object, must have a parameterized constructor.

Structures as Class Members

We have already studied that structures and classes are very similar in C++ except the default scope of members. The default scope for members of structures is *public* whereas the default visibility for class members is *private*.
Likewise, objects of different classes can act as data members, structures and unions can also act as data members of a class. In fact, all the discussion above for *Class Objects as Class Members* applies to this topic of *Structure Objects as Class Members*.

```
#include <iostream.h>
#include <stdlib.h>

struct VehicleParts
{

   int   wheels;
   int   seats;

   VehicleParts()
   {
      cout << "\n VehicleParts - default constructor";
   }

   VehicleParts(int wheels, int seats)
   {
      this->wheels   =   wheels;
      this->seats    =   seats;
      cout << "\n VehicleParts - parameterized constructor";
   }
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
   ~VehicleParts()
   {
      cout << "\n VehicleParts - destructor" << endl;
   }

} ;

class   Vehicle
{

   private :
         VehicleParts   vehicleParts ;

   public :
     Vehicle( )
      {
            cout << "\n Vehicle - default constructor" << endl;
      }
      Vehicle( int  a, int  b ) : vehicleParts( a, b )
      {
            cout << "\n Vehicle - parameterized constructor";
      }
      ~Vehicle( )
      {
            cout << "\n Vehicle - destructor";
      }
      void setPartsNum ( int a, int b )
      {
         vehicleParts.wheels = a ;
         vehicleParts.seats = b ;

      }
      void displayNumVehicleParts ( )
      {      /* The data members of the structure are public,
            therefore, directly accessible from outside. */
            cout << "\n Number of wheels for this vehicle are "
                        << vehicleParts.wheels;
            cout << "\n Number of seats for this vehicle are "
                        << vehicleParts.seats << endl;

      }

} ;

void f()
{
  Vehicle car( 4, 2 ) ;
  car.displayNumVehicleParts( ) ;
}
```

vubookshoppk.com

```
void main ( )
{
   f();
   system ( "PAUSE" ) ;
}
```

The output of the program is:

```
VehicleParts - parameterized constructor
 Vehicle - parameterized constructor
 Number of wheels for this vehicle are 4
 Number of seats for this vehicle are 2

 Vehicle - destructor
 VehicleParts - destructor
 Press any key to continue . . .
```

Classes inside Classes

In C language, structures can be defined inside structures, Similarly in C++, we can have structures or classes defined inside classes. Classes defined within other classes are called nested classes.

A nested class is written exactly in the same way as a normal class. We write its data members, member functions, constructors and destructors but no memory is allocated for a nested class unless an instance of it is created. C++ allows multiple levels of nesting. Importantly, we should be clear about the visibility of the nested class. If a class is nested inside the *public* section of a class, it is visible outside the outer (enclosed) class. If it is nested in the *private* section, it is only visible to the members of the outer class. The outer class has no special privileges with respect to the inner class. So, the inner class still has full control over the accessibility of its members by the outer class. Interestingly, the *friend* operator can be used to declare enclosed class as a friend of inner class to provide access to inner class's *private* members. This operator is used in the same way as we use it for other classes that are not nested. We can also make the inner class to access the *private* members of enclosed class by declaring the inner class as a *friend* of outer class.

The reason of nesting classes within other classes is simply to keep associated classes together for easier manipulation of the objects.

```
/* This program illustrates the nested classes */

#include <iostream.h>
#include <stdlib.h>

class Surround
{
        public :
                class FirstWithin
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
        {
            public:
                FirstWithin ()
                {
                   cout << "\n FirstWithin - default constructor";
                }
                ~FirstWithin()
                {
                   cout << "\n FirstWithin - destructor";
                }
                int getVar() const
                {
                   return (variable);
                }
            private:
                int variable;
        };

    FirstWithin  myFirstWithin;

    private:
        class SecondWithin
        {
            public:
                SecondWithin()
                {
                   cout << "\n SecondWithin - default constructor";
                 }
                ~SecondWithin()
                {
                   cout << "\n SecondWithin - destructor ";
                }
                int getVar() const
                {
                   return (variable);
                }
            private:
                int variable;
        };

// other private members of Surround

};

void f(void)
{
    Surround::SecondWithin a;
    Surround::FirstWithin b;
    Surround c;
    c.myFirstWithin.getVar();
```

vubookshoppk.com

```
}

int main()
{
    f();
    cout << endl << " ";
    system("PAUSE");
    return 0;
}
```

The output of the program is as follows:

```
SecondWithin - default constructor
 FirstWithin - default constructor
 FirstWithin - default constructor
 FirstWithin - destructor
 FirstWithin - destructor
 SecondWithin - destructor
 Press any key to continue . . .
```

Notice the access specifier (::) usage in function *f()* to access the members of inner class.

The class *FirstWithin* is visible both outside and inside *Surround*. The class *FirstWithin* has therefore global scope. The constructor *FirstWithin()* and the member function *getVar()* of the class *FirstWithin* are also globally visible. The *int variable* data member is only visible for the members of the class *FirstWithin* as it is declared *private*. Neither the members of *Surround* nor the members of *SecondWithin* can access the variable of the class *FirstWithin* directly. The class *SecondWithin* is visible only inside *Surround*. The *public* members of the class *SecondWithin* canalso be used by the members of the class *FirstWithin*, as nested classes can be considered members of their surrounding class.

The constructor *SecondWithin()* and the member function *getVar()* of the class *SecondWithin* can also only be reached by the members of *Surround* (and by the members of its nested classes). The *int variable* data member of the class *SecondWithin* is only visible to the members of the class *SecondWithin*. Neither the members of *Surround* nor the members of *FirstWithin* can access the variable of the class *SecondWithin* directly.

The nested classes can be considered members of the surrounding class, but the members of nested classes are not members of the surrounding class. So, a member of the class *Surround* may not access *FirstWithin::getVar()* directly. The nested classes are only available as type names. They do not imply as objects containment by the surrounding class. If a member of the surrounding class uses a (non-static) member of a nested class then a pointer to a nested class object or a nested class data member is defined in the surrounding class. The pointer is further used by the members of the surrounding class to access members of the nested class.

It is important to know how do we define Member functions of nested classes. They may be defined as inline functions or they can also be defined outside of their surrounding class. Consider the constructor of the class *FirstWithin* in the previous example.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

The constructor *FirstWithin()* is defined in the class *FirstWithin*, which is, in turn, defined within the class *Surround.*
Consequently, the class scopes of the two classes must be used to define a *constructor* as the following:

```
Surround :: FirstWithin :: FirstWithin ( )
{
        variable = 0 ;
}
```

The classes *FirstWithin* and *SecondWithin* are both nested within *Surround*, and can be considered members of the surrounding class. Since members of a class may directly refer to each other, members of the class *SecondWithin* can refer to *public* members of the class *FirstWithin* but they cannot access *private* members of the *FirstWithin* unless *SecondWithin* is declared as a *friend* of *FirstWithin*.
See the code snippet below, we have used *friend* operator here extensively so that all the three classes *Surround*, *FirstWithin* and *SecondWithin* can access *private* members of each other.

```
class Surround
{
        class SecondWithin ;
        public :
                class FirstWithin
                {
                        friend class Surround ;
                        friend class SecondWithin ;
                        public :
                                int getValue()
                                {
                                    Surround :: variable = SecondWithin :: variable ;
                                    return (variable);
                                }
                        private :
                                static int variable ;
                } ;
                friend class FirstWithin ;
                int getValue ( )
                {
                        FirstWithin :: variable = SecondWithin :: variable ;
                        return (variable) ;
                }
        private :
                class SecondWithin
                {
                        friend class Surround ;
                        friend class FirstWithin ;
                        public :
                        int getValue ( )
                        {
```

vubookshoppk.com

```
                    Surround::variable = FirstWithin::variable;
                    return (variable) ;
                }
                private:
                    static int variable;
            };
        friend class SecondWithin ;
        static int variable;
};
```

We can also define structures inside classes in the same manner as we defined classes within classes. Again, all the above discussion is valid for structures inside classes except the default scope of members in structures is *public* unless explicitly declared otherwise.

vubookshoppk.com

Tips

- ***A class can contain instances of other classes as its data members.***
- It is a way of reusing the code when we contain objects of our already written classes into a new class.
- The inner data members of the object are constructed and then the object itself. The order of destruction of an object is reverse to this construction order, where the outer object is destroyed first before the inner data members.
- Initializer list is used to initialize the inner objects at the construction time.
- In C++, we can have structures or classes defined inside classes. Classes defined within other classes are called nested classes.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Lecture No. 41

Reading Material

Deitel & Deitel - C++ How to Program Chapter. 12

12.2, 12.3

Summary

63) Template Functions
64) Overloading Template Functions
65) Explicitly Specifying the Type in a Template Function
66) Template Functions and Objects
67) Recap

In the previous lecture, we talked about the objects used as data members within classes. In this case, you will find that there is a lot of code reuse. A completely debug code is used as building blocks for developing new and more complex classes. Today's discussion will mainly focus on a new way of code reuse with an entirely different style of reuse. This method is called templates.

Template Functions

There are two different types of templates in C++ language i.e.' function templates and class templates. Before going ahead, it will be sagacious to know what a template is? You have used a lot of templates in the childhood. There are small scales being marketed at the stationary shops having some figures on them like circle, a square, rectangle or a triangle. We have been using these articles to draw these shapes on the paper. We put the scale on the paper and draw the lines with the pencil over that figure to get that shape. These engraved shapes are generally called stencils. But in a way, these are also templates. We may also take these 'cut-outs' as sketches. So a template is a sketch to draw some shape or figure. While drawing a special design, say of furniture, we develop a template for this, which is not an actual piece of furniture. We try that its shape should be like the outline. Later, the cut out prepared out of wood in line with the template, is actual piece of furniture. We can think of making a triangular template and then drawing it on a piece of wood and shaping it into a triangle. We can use the same template and put it on a piece of metal and can cut it into a triangle and so on. In a way, that template is allowing us the reuse of a certain shape. This is the concept we are going to try and build on here.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Here we are going to discuss the benefits of the function templates. We have been using a *swap* function. We want to interchange two things. You know the technique that we need a third temp-place holder. If we want to swap two integers *i* and *j*, the code will be as under:

```
void swap(int &i, int &j)
{
        int tmp;
        tmp = i;
        i = j;
        j = tmp;
}
```

This is a very generic way of interchanging two values. We have written a swap function to interchange two integers. To interchange two doubles, we have to come up with some other swap function for doubles and so on. Whenever, a need to use this swapping technique for different data type arises, we have to write a new function. Can we write such functions? Yes, we can. These functions can be overloaded. We can have functions with the same name as long as the types or the number or the arguments are different. Compiler can detect which function should be used. It will call that function appropriately. So you can define *swap* for integers, floats and doubles. There is also no problem in defining multiple versions of this function with different data types. Depending on what is required, the compiler will automatically make a call to the correct function. This is the overloading. The code for every data type looks like:

```
void swap(SomeDataType &firstThing, SomeDataType &secondThing)
{
        SomeDataType tmp;
        tmp = firstThing;
        firstThing = secondThing;
        secondThing = tmp;
}
```

This is a sort of generic code, we are writing again and again for different data types. It will be very nice if somehow we can write the code once and let the compiler or language handle everything else. This way of writing is called templates or function templates. As seen in the example of a template of a triangle, we will define a generic function. Once it is defined and determined where it will be called for some specific data type, the compiler will automatically call that function.

As discussed in the example of overloaded functions, the automatic part is also there. But we wrote all those functions separately. Here the automatic part is even deeper. In other words, we write one template function without specifying a data type. If it is to be called for *int* data type, the compiler will itself write an *int* version of that function. If it is to be called for double, the compiler will itself write it. This does not happen at run time, but at compile time. The compiler will analyze the program and see for which data type, the template function has been called. According to this, it will get the template and write a function for that data type.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Now, we will see the idea or technique for defining template function. Here you will come across some new keywords. First keyword is "template". These are the recent addition to the language. Some old compilers may not have these features. However, now almost all of the compilers implement these features. Another keyword is *class* that is quite different than we have been using for defining the classes. This is another use of the same keyword. Normally, when we define a generic function, it is independent of the data type. The data type will be defined later in the program on calling this function. The first line will be as *template<generic data type>*. This generic data type is written while using the *class* key word as *template<class variable_name>*. So the first line will be as;

```
template<class T>
```

We generally use the variable name as *T* (T evolves from template). However, it is not something hard and fast. After the variable name, we start writing the function definition. The function arguments must contain at least one generic data type. Normal function declaration is:

```
return_type    function_name(argument_list)
```

return_type can also be of generic type. There should be at least an argument of generic type in the *argument_list*. Let's take a very simple example. This is the function *reverse*. It takes one argument and returns its minus version. The *int* version of this function is as:

```
int reverse(int x)
{
        return (-x);
}
```

Similarly its double version will be as:

```
double reverse(double x)
{
        return (-x);
}
```

Similarly, we can define it for other data types.
Let's see how can we make a template of this function. The code is as:

```
template<class T>

T reverse(T x)
{
  return (-x);
}
```

In the above function definition, we have used *T* as generic type. The return type of the function is *T* which is accepting an argument of type *T*. In the body of the function, we just minus it and return *x* that is of type *T*. Now in the main program, if

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

we write it as *int i* and then call *reverse(i), what* will happen? The compiler will automatically detect that *i* is an *int* and *reverse* is a template function. So, an *int* version of *reverse* function is needed in the program. It uses the template to generate an *int* version. How does it do that? It replaces the *T* with *int* in the template function. You will get exactly the same function as we have written before as *int reverse(int x).* This copy is generated at compile time. After the compilation, all the code is included in the program. A normal function call will happen. When we write *reverse(i)* in the main or some other function, it is not required to tell that an *int* version is needed. The compiler will automatically detect the data type and create a copy of the function of the appropriate data type. This is important to understand. Similarly if we have *double y;* and we call the *reverse* function as *reverse(y)*; the compiler will automatically detect that this program is calling *reverse(i)* and *reverse(y). Here i* is an *int* and in *reverse(y), y* is a *double*. So the compiler will generate two versions of *reverse* function, one with *int* and the other with *double*. Then it will be compiled and the program will execute correctly. This is the classic example of code reuse. We have to pay attention to writing the template. It should be generic in nature.

For a programmer, there are facilities of the macros and *#define* which have the limitations. Macro is a code substitution while *#define* is a value substitution. Here, in templates, we write a generic code and the compiler generates its copies of appropriate types. It is always better than ordinary function overloading. Now let's take the previous example of *reverse*. When we write the function of *reverse* and give it a value of type *double*, a version of the *reverse* function for *double* is created, compiled and used. If we write the same template in some other program and call it for an integer, it will still work. It will automatically generate code for *int*. We should write a template while doing the same functionality with different data types. The rule for templates is that at least one argument in the function should be of generic data type. Other wise, it is not a template function. We write a template *class T* and use *T* as a new data type. Being a template data type, it does not really exist. The compiler will substitute it on its use besides generating an appropriate code. There are some limitations that should be kept in mind. We cannot store the declarations and definitions of these functions in different files. In classes, we have this for certain purposes. In case of a class, we put the declaration of the class and its basic structure in the header file to facilitate the users to know what the class does implement. The definition of the class, the actual code of its functions and the manipulations are provided along with as object code. Here in the template case, the compiler makes a copy of the source code and converts it to object code. We cannot give the declaration of the template function in one file and the definition in some other. If we store these in different files, it will not compile. It does not have real data type and still has parameterized or generic data type in it. So the declaration and definition of a template function should be in the same file. We will include this file or keep the template with our main program. When it will be used, the copies of code will be automatically generated. So it is a slight limitation with templates. In any case, template class or template functions are for our own use. We do not write template functions as libraries for other people as it is like giving away our source code.

For template functions, we must have at least one generic argument. There may be more than one generic arguments. We have to pass it to pieces of data to be swapped. We can write swap function as:

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
template<class T>

 void swap(T &x, T &y)
{
                T tmp;
                tmp = x;
                x = y;
                y = tmp;
}
```

In the above function, we are passing both arguments of generic type and declared a *tmp* variable of generic type. We can also mix generic data types and native data types including user defined data types. This template version of swap function can be used for integer, double, float or char. Its copy will be created after writing the *swap(a, b)* in the program. If we have *int a, b;* in the program, *int* copy of *swap* function will be generated. If you have written *char a, b;* a *char* copy of *swap* function will be generated. A copy is simple substitution of *char* with *T*. Just replace *T* with *char* in the *swap* function and remove the first line i.e. *template<class T>*, this is the function that the compiler will generate. Now we have seen examples of one generic and two generic arguments functions. You can write template functions with more than two generic arguments.

So far, we have been using only one generic data type. However, the things can not be restricted to only one generic data type. We can use more than one generic data types in template functions. We can do that by extending the *template<class T>*. The use of two generic types can be written as:

```
template <class T, class U>
```

We can use any name in place of *T* and *U*. Two data types can be mixed here. So we can write a function that takes an *int* and *float* and can multiply these two. We can use *T* as *int* and *U* as *float* or whatever is the function requirement. Let's look at another example of template function. We want to write a function that takes two arguments of same type and tells which of the two is greater. The code will be as below:

```
// A small program shows the use of template function
#include<iostream.h>

// template function of deciding a larger number
template<class T>
T larger(T x, T y)
{
   T big;
   if (x > y)
     big = x;
   else
     big = y;
   return(big);
}
```

vubookshoppk.com

```
// the main function
void main()
{
   int i = 7, j = 12;
   double x = 4.5, y = 1.3;
   cout << "The larger of " << i << " and " << j << " is " << larger(i, j)<< endl;
   cout << "The larger of " << x << " and " << y << " is " << larger(x, y)<< endl;
   //cout << "The larger of " << x << " and " << y << " is " << larger(i, y)<< endl;

}
```

The output of the program is:

```
The larger of 7 and 12 is 12
The larger of 4.5 and 1.3 is 4.5
```

The function *larger* is very simple. We have two arguments in it, compared these two arguments and set the variable *bigger*. You have noticed that the definition of *larger* is not exactly correct. In the *if* condition, we check that *x* is greater than *y*. So in the else-part *x* can be equal to or lesser than *y*. Let's see their use in the main. We declare two integers *i* and *j* and two doubles *x* and *y*. Then we use the *cout* to display the result. The *larger* function will return the bigger argument. When we write *larger(i, j)*, the compiler will detect it and generate an *int* version of the larger function. In the next line, we have used *larger(x, y)* as *x* and *y* are double. Here, the compiler will generate a double version of the *larger* function. Now compile the program and it is ready to be executed. The two versions of *larger* functions will be executed .We get the larger of two integers and larger of two doubles. You have noticed that the last line of the code is commented out. In that line, we are trying to call the *larger (i, y)*. There is a problem that if you uncomment this line, it will not be compiled. We have only defined one generic class type in the templatized function i.e. *class T*. Here we are trying to call it with an *int* and a *double*. The compiler does not know what to do with it. Either it should promote the *int* to *double* and call the *double* version or demote the *double* into *int* and call the *int* version. But the compiler will not make this decision. It will be a compile time error. We can write another template function that handles two data types or try to call this function with one data type. Be careful about these fine points. Template is a nice thing to use but needs to be used carefully. The compiler may give an error, so you have to correctly use it. Here in the larger function, we have to provide both arguments of the same data type.

Following is an example of *larger* function using two different generic types.

```
// A template function example using two generic types
#include<iostream.h>

// template function
template <class T, class U>
void larger(T val1, U val2)
{
  if (val1 > val2)
     cout<<"First is larger"<<endl;
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
  else
     cout<<"First is not larger"<<endl;
}

// main function
void main()
{
  larger(2.1, 9);
  larger('G', 'A');
}
```

The output of the program is:

```
First is not larger
First is larger
```

Overloading Template Functions

Let's take benefit of our knowledge and discuss the things of the next level i.e. function overloading. Under the techniques employed in function overloading, the functions have the same name but differ either by the number of arguments or the type of the arguments. Remember that the return type is not a differentiator when you are overloading the functions. Now if the number or type of the arguments is different and the function name is same, the compiler will automatically call the correct version. The same rule applies to the template function. We can write overloaded template functions as long as there is use of different number or type of arguments.

We have written a templatized *swap* function. Let's rename that function as *inverse*. It will swap the variables. We have another *inverse* function that takes one argument and return the minus of the argument supplied. We have two template functions named *inverse*. Here is the code of the program:

```
//  An example of overloaded template functions.

#include<iostream.h>

// template function
template<class T>
void inverse(T &x, T &y)
{
   T temp;
    temp = x;
    x = y;
    y = temp;
}

// overloaded inverse fucntion
template<class T>
T inverse(T x)
{
  return (-x);
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
}

// the main fucntion
void main()
{
  int i = 3, j = 5;
  // calling the templatized functions
  inverse(i);
  inverse(i, j);
  cout << "i = " << i << ", j = " << j << endl;
}
```

The output of the program is:

```
i = 5.6, j = -3.4
```

In the above program, we have overloaded template functions. When we write *invers(i)*, the compiler will detect the *inverse* function with one argument and generate its *int* code. However, on writing *inverse (i, j)*, it will generate an *int* version of the *inverse* function which takes two parameters. This is not a good example as the function names are confusing. The function which does swapping should be named as swap while the one doing negative should be named as negative. There might be good occasions where you might want to use overloaded templates. The same rule of ordinary function overloading applies on template function overloading.

Explicitly Specifying the Type in a Template Function

In the template functions, sometimes we want to see which version of the template function should be used. Let's take the example of *reverse* function. We call that function for *double* data type. A function for the *double* would have been generated and its negative value will be returned. Suppose we want to pass it a *double* but return an *integer*. We want to return a negative integer that was a part of the double variable, passed to the function. We can force the compiler to generate an *int* version of this function while not passing it an *int*. It can take place when we are going to call the function in the program. We write the data type in the angle brackets between the function name and argument list. For example, if we have a template *reverse* function, which returns *-x*. In the program, we have *double a*. As soon as we write *reverse(a)*, the compiler will generate a *double* version of *reverse* function. Now we want that 'a' should be passed to this function while returning an *int*. The prototype of the function is *T reverse(T x)*. We want that *T* should be replaced by *int*. At the same time, we want to pass it *double*. To obtain this, we will write as *reverse <int> (a);* writing *<int>* forces the compiler to also generate an integer version of the function. There may be instances where this technique is useful.

Suppose, we have a template of *reverse* function that depends on two generic data types. The function template is as follows:

```
template <class T, class U>

T reverse (U x)
{
        return -x;
```

}

Now the return type is *T* while the argument is of type *U*. In the body of the function, we return *-x* and the conversion automatically takes place. In this function template, we are using two generic types. The return type cannot force anything as it is used later in the assignment statement. If we have *double a* in the program, and say *reverse(a);* What version will be generated? What will be replaced with *T* and *U*? We can force it as *reverse<int>(a);* In that case, it will force *T* to become *int*. It will force *U* to become of the type *a* i.e. *double*. It will take a double number, reverse and convert it into *int* and return it. You can explicitly specify it as *reverse<int, double> (a);* so we have specified both *T* and *U*. We are specifying this to the compiler so that when the compiler generates the code, it carries out it for these versions. It is like the default argument list. You can not force the second part only i.e. you can not force *U* only while missing *T*. It has to go left to right. You can do as *revere<double, double> (a)*; or *reverse(double, int>(a)*. The appropriate versions will be generated. Actually, you can force what type of versions should be generated in your code. Normally, we do not need to force the template functions. Normally the template function is used for different data types while generating appropriate versions by the compiler.

Here is the code of the above-explained program:

```
// An example of forcing the template functions for some specific data type
#include<iostream.h>

template <class T, class U>
T reverse (U x)
{
  return (-x);
}

// main function
void main()
{
   double amount = -8.8;
   // calling the function as double reverse(int)
  cout << reverse<double, int>(amount) << endl;

   // calling the function as double reverse(double a)
   cout << reverse<double>(amount) << endl;

   // calling the function as double reverse(double a)
   cout << reverse<double, double>(amount) << endl;

   // calling the function as int reverse(int a)
   cout << reverse<int, int>(amount) << endl;
}
```

The output of the code is as follows:

```
8
8.8
```

vubookshoppk.com

```
8.8
8
```

Template Functions and Objects

We have seen the template functions and know that classes also have member functions. Can we use these templates with the member functions of a class? Yes, we can templatize member functions. The operations used within template functions should be present in the public part of the class. Let's see an example to understand this. Suppose we have created a class *PhoneCall*. We have a *lengthOfCall* data member in the class that tells about the duration of the call. Another character data member is *billCode*. The *billCode* will tell us that this call is local, domestic or international. Suppose, we browse the bill and notice a wrong call. What should we do? We will pick up the phone and call the phone company or go the phone company office to get the bill corrected. How will they do that? They will verify it with the record and see there is no such call or the duration is not chargeable. So far, we have been using the *reverse* function to minus the input argument. Here we want to reverse the phone call. Suppose, we define that when the *billCode* is 'c', it means that call has been reversed or cancelled. Can we use this concept and write a *reverse* function for this class. Let's revisit the *reverse* function template.

```
template<class T>

T reverse(T x)
{
  return (-x);
}
```

vubookshoppk.com

Here *T* is the generic type that will be replaced with int, float etc. Can we replace *T* with the object of the class *PhoneCall*. How will that work? Let's replace the *T* with *PhoneCall*, the code looks like:

```
PhoneCall reverse(PhoneCall x)
{
  return (-x);
}
```

The declaration line shows that it returns an object of *PhoneCall* and takes an argument of type *PhoneCall*. Inside the body of the function, we are returning *-x*. What does *–PhoneCall* mean? When we are using template functions in the classes, it is necessary to make sure that whatever usage we are implementing inside the template function, the class should support it. Here we want to write *–PhoneCall. So* a minus operator should be defined for the *PhoneCall* class. We know how to define operators for classes. Here the minus operator for *PhoneCall* will change the *billCode* to 'c' and return the object of type *PhoneCall*. Let's have a look on the code.

```
// A simple program to show the usage of the template functions in a class

#include<iostream.h>

// reverse template function
```

```
template<class T>
T reverse(T x)
{
  return (-x);
}

// definition of a class
class PhoneCall
{
  private:
          int lengthOfCall; // duration of the call
          char billCode;   // c for cancelled, d for domestic, i for international, l for local

  public:
    PhoneCall(const int l, const char b); // constructor
    PhoneCall PhoneCall::operator-(void); // overloaded operator
    int getLengthOfCall(){ return lengthOfCall;}
         void showCall(void);
};

PhoneCall::PhoneCall(const int len=0, const char b='l')
{
          lengthOfCall = len;
          billCode = b;
}

void PhoneCall::showCall(void)
{
   cout <<"The duration of the call is " << lengthOfCall << endl;
   cout <<"The code of the call is " << billCode << endl;
}
// overloaded operator
PhoneCall PhoneCall::operator-(void)
{
   PhoneCall::billCode='c';
   Return (*this);
}

// main function
void main()
{
          PhoneCall aCall(10, 'd');
          aCall.showCall();
          aCall = reverse(aCall);
          aCall.showCall();
}
```

The output of the code is:

```
The duration of the call is 10
The code of the call is d
```

vubookshoppk.com

```
The duration of the call is 10
The code of the call is c
```

We have overloaded the minus operator in the *PhoneCall* class. We cannot change the unary or binary nature of operators. The minus operator is lucky due to being unary as well as binary simultaneously. Here, we have overloaded unary minus operator so, it can be written as *-x*. The definition of operators is same as ordinary functions. We can write whatever is required. Here we are not taking negative of anything but using it in actual meaning that is reversing a phone call. In the definition of the minus operator, we have changed the *billCode* to 'c' that is cancelled and the object of type *PhoneCall* is returned. Now look at the definition of the *reverse* template function and replace the *T* with *PhoneCall*. In the body of the function where we have written *–x*, the minus operator of the *PhoneCall* class will be called. Since it is a member operator and the calling object is available to it. Now let's look at the main program. We take an object of *PhoneCall* as *PhoneCall aCall(10, 'd');* The object *aCall* is initialized through the constructor. Now we display it as *aCall.showCall();* that shows the length of the call and bill code. After this, we say *reverse(aCall);* The reverse function should not be changing *aCall* and should return an object of type *PhoneCall*. It means that *aCall = reverse(aCall);* the object returned is assigned to *aCall*. Reverse is a template function, so the compiler will generate a *reverse* function for *PhoneCall*. When it will call *-x*, the member minus operator of the class *PhoneCall* will be called. It will change the *billCode* of that object and return the object. As we have written *aCall = reverse(aCall)* so the object returned from *reverse* having *billCode* as 'c' will be assigned to *aCall*. Now while displaying it using the *aCall.showCall()*, you will see that the *billCode* has been changed. So reverse works.

Recap

Let's just recap what we just did in this lecture. We have defined a template function *reverse*. In the template definition, this function returns *-x* whatever *x* is passed to it. After this, we wrote a *PhoneCall* class and defined its minus operator. Whenever we have to take minus of the *PhoneCall*, this operator will be called. This action is based on the phone call domain and is to change the bill code to 'c'. So the minus operator returns an object of type *PhoneCall* after changing its bill code. Now these two are independent exercises. The class *PhoneCall* does not know about the *reverse* function. It only has defined its minus operator. On the other hand, the *reverse* template function has nothing to do with the *PhoneCall* class. It is the main program, linking these two things. The main function declared an object of type *PhoneCall* and called the *reverse* function with that object. When we write the statement *aCall = reverse (aCall);* the compiler automatically detects that we have got a situation where *reverse* template function is called with the object of type *PhoneCall*. It needs to generate a copy of this template function that will work with *PhoneCall*. When it goes to generate that copy, it encounters with *return(-x)*. It has to know that a minus operator exists for that class. If we have not defined the minus operator what will happen. The compiler may give an error that it does not know what is *-PhoneCall*. On the other hand, sometimes default substitution takes place. It may not be what you want to do. You have to be careful and look at the implications of using a template function with a class and make sure all the operations within template function should be defined explicitly for the class so that function should work correctly. Once this is done, you realize that life has become simpler. The same *reverse* function works for this class. Now you can extend the concept and say how to reverse a car, how to

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

reverse a phone call, how to reverse an int and so on. The idea is combining two very powerful techniques i.e. operator overloading and template mechanism which provides for writing the code at once. In the normal overloading, the facility is that we can use the same name again and again. But we have to write the code each time. Normally, the code is different in these overloaded functions. In this case, we are saying that we have to write identical code i.e. to reverse something, swap two things. So the code is same only data type is different then we should go and define a template for that function and thus, template is used again and again. We started with the template function, used at program level. The use of template with class was also demonstrated. This combination has some rules. This is that all the operations that template function is using should be defined in the class. Otherwise, you will have problems.

vubookshoppk.com

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Lecture No. 42

Reading Material

Deitel & Deitel - C++ How to Program Chapter. 12, 20

12.4, 12.5, 12.7, 12.8, 20.1

Summary

68) Class Templates
69) Class Templates and Nontype Parameters
70) Templates and Static Members
71) Templates and Friend Functions
72) Example
73) Sample Program
74) Advantages and Disadvantages of Templates
75) Standard Template Library

VUbookshoppk.com

As discussed earlier, template functions are utilized while writing functions for generic data type. We take benefit of templates in case of writing the same function repeatedly. In this case, the writing code seems very similar, in fact identical. But the data type is changed for different versions. So we write a function for generic data type whose syntax we have used as under

template <class T>

Here T is a generic data type. Now in the function we write T while dealing with a data type. This becomes a generic template of the function. During the process of using this template, this function with a particular data type is called. The compiler automatically detects the type of the data passed (say we passed an int) and generates a new copy of the function with *int*. Here T is written in the original template. The copy is compiled to the object code while existing in the program. The same thing applies to other data types, used for the same function. Thus, we create a family of functions with a single template. The functionality of these functions is the same but with different data types. For example, if we want to find the square of a number, a template square will be written first. It will be called with int, double or float. Otherwise, we have to write the over loaded versions of the square function for different data types. So template functions are of good use for the programmers. They promote code reuse. The major advantage of their use is that once we write correct code with correct logic for a template function, there will be no need to re-write it.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

How can we test a template? We write the template in reverse technique at the moment. When it is known what the template has to do, we take a data type for example *int*. A complete function for int is written and tested. After the ascertainment of the accuracy of function's working, the *int* in the function is replaced with T and declared a template by writing *template <class T>*. We cannot see the code generated by the compiler in the editor. So it becomes difficult to debug the template code. We have to read and check the code carefully while writing it.
After having a detailed discussion on the template function, we will now talk about template classes.

Class Templates

Creation of a data type of our own with the same behavior for *int*, *float* and *double* etc, is the case of defining a complete interface and implementation in a generic fashion. To further understand this concept, let's talk about a data structure called stack. We will now try to understand how does stack work, what its properties are and can we make it generic.
You might have seen the plates, kept together in a orderly manner i.e. one on the other. This is a stack. Now if someone wants to add a plate on the pile, he will have to put it on the top of the stack. So, there is only one way to add something on the stack. Similarly, if we want to pick a plate from the pile, it will be taken from the upper-most tier. Thus the property of the stack is that the last placed thing will be picked first. This phenomenon is called 'Last-in, first out' or LIFO. In programming, we can understand what thing we want to add, the required thing is added to the top of the stack. When we pick up a thing from it, the last placed item is picked first. Following this rule of stack (last in first out), we can make a stack of integers, floats and doubles etc. Here, the stack is a class with a defined behavior, interface and the data, it holds. Now we say that the data held by the class is variable to help make a stack of integers, floats or doubles. Thus, stack is a good candidate for a template class. It means that when we instantiate the class for creating objects, a stack of integers or floats is required. The behavior of the compiler in template classes is the same as in template functions. If we want to instantiate a template class with a new data type, the compiler will generate a new version of the class with the specific data type at the place of T in the template class.
We know that a class is a user-defined data type. With the help of a template class, we make another class of the user defined data type. In other words, things are not restricted to creating copies of class only for native data type. Copies of class of our own data type can also be created. It is a case of a real extensibility.
Let's see the syntax of this generic template class. It is similar to the simple template function in which we write *template <class T>*. Here T is the placeholder that will be replaced by the data type when we use it. The syntax of the template class is

```
template <class T>
class class-name()
{
        definition of class
};
```

In the definition of the class where the generic data type is required, we write T. For example, there is a class in which we want to write *int* data type. The *int* is the data

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

type that may be a *float* or *double* at different times. For this, T is written wherever we are using *int* in the class definition. Be careful that some times, we may use integers as it in a class. For example, when we create a vector or array, its size will be an integer. But the array will be of integers, floats or chars. So don't confuse it them. It is better to replace T whenever necessary.
We start writing of a template with the following line

```
template <class T>
```

Later, definition of the class in ordinary fashion begins. We should use T wherever in case of employing the generic data type. T is not something fixed for this purpose. We can use a, b or c or whatever needed. However, T is normally used.
The member functions are normally defined out side the class. To define the member functions of the template class, we write

```
template <class T>
class name <T>::function name (argument list)
{
        // function body
}
```

In the function body, the programmer will write T wherever it is needed. This way, the template-class and its member functions are defined. However, when we use the class in main program or other function, the objects of this class are declared. Suppose there is a class Number, say *Number x*; As Number is a template class, we will have to tell the type of the number. Let's see the code of this simple template class Number. The Number class can store and display a number. The definition of the class is written as under.

```
template<class T>
class Number
{
        private:
                T myNumber;
        public:
                Number( T n );
                display();
};
```

We create an object of this class in main program by telling the type of the number in the following way

```
Number <data type>
```

Here data type may be *int*, *float* or *double*. Now we can create an object of this class for an integer type as follows.

```
Number <int> x ;
```

We can read the line as *x* is an object of class Number and its parameter is an *int*. The way of analyzing it is that wherever we wrote T in the class, now *int* is written there.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Compiler does it automatically. We will not see the code of this class written for *int*. The compiler generates a copy of the class for *int* and uses it. Similarly, if we write

Number <double> y ;

Here *y* will be an object with the data member of type *double*. Again, the entire copy of the class will be created with *double* everywhere instead of T. The program will compile and run. So it is quiet useful and a big shortcut.

Class Templates and Nontype Parameters

There is a little variation, which is we can also use non-type parameters in templates. What do non-type parameters mean? We have been writing *template <class T>*. However, while writing *template <class T, int element>*, the non-generic type (i.e. int) will be treated as a constant in the class definition. In the class definition, wherever we use the name *element*, it will be replaced by the value passed to it. Arrays when declared and given a number in square brackets, the number will be a constant. Similarly, while using with the # sign, we associate a name with a number which is a constant. Here the non-type parameter in a way behaves like a constant. We can use it to give a dimension to the things. Instantiating a class, we not only replace T but also provide a value for the non-type parameter defined in the class definition.
By using templates, we save a lot of effort and labor. The other big motivating factor is the reuse of tested and tried code for a particular problem.

Templates and Static Members

Now let's talk about the implications of the template classes. To understand the behavior of templates with static members, we have to comprehend the concept of static member variables. Static variable members are used to define ordinary classes. The static variable has a single copy for the whole class. So there are not separate copies of the static data variable for each object like ordinary data members.
Now let's see what happens when a static member is a part of a template class. The instantiation of the class has two parts i.e. one is creating an object while the other is the type of the object. For example, from the previous class *Number*, we write

Number <int> x ;

Here *x* is an object of generic class Number with a specific type *int*. We can also use *float* or *double* instead of *int*. We suppose, there is a static variable in the *Number* class. On instantiating the class with *int*, there will be a copy of static variable for *int* set-off objects. If we instantiate the class for *float*, there is going to be a copy of the static member for float numbers. So the member is static (i.e. there is one copy) for that type of the objects. There will be one static value for different object created with type *int* while another static value for different objects created for type *double*. So, this static value is not class wide. It is something of specific nature. In simple words, the compiler reads the code of program (main or other function) and generates a copy of the template class accordingly. It also gives a name of its own to this copy. Thus in

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

a way, the compiler generates a new unique class, replacing T with *int* (or any other data type we want). The static member of this unique class behaves exactly like the ordinary class. Similarly the compiler generates a copy for *double* with a unique name. Here the static member of this copy will affect the objects created for *double* data type. The static variables are instantiated once for each type whenever we instantiate a class while replacing generic data type with a specific data type.

It is pertinent to note that we can replace the generic data type with our own data type. This is slightly tricky. Suppose, we have written a class, '*Person'*. There is also a generic class Array, which can be instantiated with *int*, *float* or *double* data type that means it may be an array of integers, floats and doubles respectively. Can we do so with an array of persons? If we have defined a class called *Person*, there may be an array of *Person*. *Person* now behaves like another data type. At the moment, it does not matter whether the data type is user defined or not.
We have to be careful that when we are using our own object i.e. our own class in a template, it must support the functions and interfaces, needed for this generic structure of the class. So don't put in something that cannot be used by this generic structure. We have discussed an example of phoneCall where *reverse* returns *x* by converting it to *–x*. In that example, we had to define the minus (-) operator for phone call. Similarly, in that example, billCode is changed to 'c'. If number is passed, the negative number will be returned. Its behavior was changed in phoneCall. So we have to take care of these things.
Whenever we use a template class and instantiate it for one of our own classes, it is necessary to have compatible function calls in it. It means that member functions behave properly as per requirements.

Templates and Friend Functions

Now we will have a look on another concept i.e. friend functions. We have read in classes that a programmer can declare functions as friend of the class. Let's first look at the need of friend functions. Suppose we have defined an operator for our class, say + operator. We know that + is a binary operator that takes two arguments. It implements a + b. While implementing + operator for a class, we see that the calling object is on the left side of the operator (i.e. a). The + operator gets this calling object through *this* pointer. It has an argument on the right hand side i.e. *b*. Here, *a* is an object of our class and *b,* an ordinary data type. Similarly, we have *a + 2 ;* where *a* is an object of a class and 2, an ordinary *int*. Now this + operator has to behave intelligently. This way, we have over loaded it within the class definition. What happens when we say *2 + a ;* ? In case of ordinary integers, 2 + 3 is same as 3 + 2. So we want the same behavior in a class. We want *2 + a* behaving the same way as *a+ 2*. We cannot carry out overloading of a member function for this. When we write *2 + a;* there will be an *int* on left- hand side of the + operator. It is not a member function of the class. For a member function or member operator, the object on left- hand side should be that of the class. So if we want *2 + a* ; we have to write a friend function for it. Here, the private data of the class is manipulated by this function. For this purpose, there is need to have access of this function to the private data of the class. The only way through which an external function can have access to the private data of the class is declaring that function to be a friend of the class. So this was the motivation of the friend function.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Now what will be the behavior of friend functions in a template class. For example if we write in our template class

friend f ();

Here *f* is function name. Thus in above statement, we say that *f* is a friend function of the class. It is declared in a template class. While creating an object of a template class, we tell the type (*int*, *float or user defined data type* etc) of which the class is to be generated. Now when we have written *friend f()*, *f* becomes a friend function for all classes generated by using this template. So it is very global that means *f* will have an access to the private data structure of any and all different classes which are instantiated by this template class.
Now we write T in the argument list of the friend function. If we have instantiated a class for an integer, and we have written the friend function *f* with T as *f* <*int*>. This will mean that this function is a friend for all *int* versions of this class. This is an interesting concept. If we have a *double* version of the class, this *f* (i.e. f<int>) will not be a friend of it. It is only a friend of the integer versions of the class.
Similarly, if we have a friend class declared in the template class as

friend class Y;

then it means that all the member functions of the class *Y* can access the private data of any class-type generated with the help of this template. For example, if we generate a class for *int*, the member functions of class *Y* can handle the data structure of the object of this class of type *int*. If we instantiate a class for *double* from this template, the member functions of *Y* can handle the data of the object with *double* version.
Similarly if we write in a template class

friend A :: f ()

Here *f* is a function. It means that the member function *f* of a class *A* is a friend of this template class. We have not granted access to the whole class, but only to a function of that class. That access applies to all classes generated using this template.
Finally, if we use <T> with the function *f*, it becomes specific. In other words, this friend function (i.e. written with <T>) will be a friend of classes generated by the template with T data type. It will not be a friend of the other versions of the class. Here T may be *int*, *float*, *double* or any user defined data type.

Example

Let's apply these concepts on one specific example. We may create a class called *Stack* in a generic fashion. There are some properties of the stack. Firstly, we should be able to know that at what position in the stack we are at a particular time. So this is a concept of stack pointer. We take an array for stack. The stack pointer always points to the top of the stack. It will be good to make the array generic so that we can make an array of any data type. Then there are few questions like is stack empty or full etc. Here the code seems fairly straight- forward. We start with

template class <T>

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

and then write

class Stack

In the class definition, An integer variable called *size* is declared for the size of the array. Then we declare the array and write its data type as T, which is a generic type .It will be replaced by *int*, *float*, *double* or *char* when we will use this array. We can use the dynamic memory allocation for the array. But we use a fixed size array for the sake of simplicity. To declare an array, we need a constant value. Normally, this constant value is not written in the class definition. It will go to the constructor and be required when the constructor will be called for an object. We can use the constructor to actually define the array for us. We need some utility functions. The function *push()* is used to push an element on the stack. We use the function *pop()* to get an element from the stack. The *push()* and *pop()* functions put and get things of type T. So *pop()* should return something of type T. That means it will return *int* if *int* is pushed and returns *double* if *double* is pushed and so on. So we need *push()* and *pop()* which are parameterized with T. After this, there is need of some functions for generic manipulation like if stack is full or if stack is empty. We can write function *isempty() that* returns a Boolean. If it returns TRUE, the stack will be empty. However, presence of something in the stack will turn it FALSE. Similarly we can write a utility function *isfull()* to check whether the stack is full. We cannot store elements more than that size in the stack. The *isfull()* returns TRUE, if the stack is full. The code of the class definition is very simple. We write T wherever we need a generic data type. It can be written as under.

```
template <class T>
class Stack
{
        private :
                int size ;
                T array [ ] ;
        public :
                Stack ( ) ;
                void push ( T ) ;
                T pop ( ) ;
                bool isEmpty ( ) ;
                bool isFull ( )
} ;
```

In the definition of the functions of the class, we again use <T> immediately after the name of the class. It will be followed by the resolution operator (::) and the function name and finally we write T, wherever we want to use generic data type. It's a definition of the class Stack. While using it, say for *int*, we write *Stack* <*int*> and provide a initializer value so that it can determine the size of the array. We read it as 'create a stack of type int'. Similarly *Stack*<*double*>*x* will mean *x* is a stack of type *double*. The main advantage of this process is that we write the *Stack* class once as the behavior is common regardless of the type of the data we want to put on. Stack class can be used for *int*, *double* or *char* data type.

This is the analysis of the example of Stack class, now as a programmer, it is left to you to write the complete code of the example.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Sample Program

Here is a sample program that demonstrates the use of template class.

```
/* This program defines a template class and shows its use for different data types.
There is also the use of template function. It also overloads the << operator.
*/
#include<iostream.h>

template<class T>
class Generic
{
    private:
        T instance;
    public:
        Generic(T i);
        void print(void);
};

//generic constructor
template<class T>
Generic<T>::Generic(T i=0)
{
        instance=i;
}

template<class T>
void Generic<T>::print(void)
{
        cout<<"Generic printing: "<<endl;
        cout<<instance<<endl;
}

class Employee
{
   private:
        int idNum;
        double salary;
   public:
        Employee(int id);
        friend ostream& operator <<(ostream& out, const Employee &e);
};

Employee::Employee(int id=0)
{
        idNum=id;
        salary=4.9;
}

ostream& operator<<(ostream &out, const Employee &emp)
```

vubookshoppk.com

```
{
        out<<"Employee number "<<emp.idNum;
        out<<" Salary "<<emp.salary;
        return(out);
}

void main()
{
        Generic<int>anInt(7);
        Generic<double>someMoney(6.65);
        Generic<Employee> aWorker(333);
        anInt.print();
        someMoney.print();
        aWorker.print();
}
```

Following is the output of the program.

```
Generic printing:
7
Generic printing:
6.65
Generic printing:
Employee number 333 Salary 4.9
```

Advantages and Disadvantages of Templates

Although, most of the following uses can also be implemented without templates; templates do offer several clear advantages not offered by any other techniques:

- Templates are easier to write than writing several versions of your similar code for different types. You create only one generic version of your class or function instead of manually creating specializations.
- Templates can be easier to understand, since they can provide a straightforward way of abstracting type information.
- Templates are type-safe. This is because the types that templates act upon are known at compile time, so the compiler can perform type checking before errors occur.
- Templates help in utilizing compiler optimizations to the extreme.

Then of course there is room for misuse of the templates. On one hand they provide an excellent mechanism to create specific type-safe classes from a generic definition with little overhead. On the other hand, if misused

- Templates can make code difficult to read and follow depending upon coding style.
- They can present seriously confusing syntactical problems esp. when the code is large and spread over several header and source files.
- Then, there are times, when templates can "excellently" produce nearly meaningless compiler errors thus requiring extra care to enforce syntactical

vubookshoppk.com

and other design constraints. A common mistake is the angle bracket problem.

Standard Template Library (STL)

Templates are a major code reuse feature. History of C++ language reveals that the template feature was introduced later, relative to other features. But it is a very important feature. We will realize that it makes a lot more sense to keep total code base very small and very concise. It also helps ensure that the same tested code is used everywhere. We had earlier referred to this concept while writing classes. We separated the interface and implementation and sealed the implementation after testing it. Afterwards, we created different objects of the class and every object knew its behavior. Thus there was an abstraction of details. The template functions and template classes go one-step even further. With templates, we can perform different tasks while using one base code. Objects of different types staying with one particular framework can be instantiated. This framework (template) is so important that a couple of researchers actually sat down and started looking at that in programming we often are using the one concept which applies to so many things that we should templatise it. For example, with the help of arrays, we do different manipulations like, 'next element', go to the end of the array, add something at the end etc. Now suppose that the size of array is 100. We want to add the 101st element in the array. We can do it by copying the same array in a new big array and adding the element to that array. Thus we have solutions for different problems, but these are the things of very common use. Their every day use is so important that two researchers wrote a whole library of common use functions. This library is a part of the official standard of C++. It is called STL i.e. Standard Template Library. As a library, it is a tested code base. Some one has written, tested and compiled for the ultimate use of programmers. We can use these templates and can implement different concepts for our own data types. Equally is true about the use of the array data type. Our code will become very small with the use of this tested facility. Similarly, there is no bug or error in it. Thus, if we have a tested and tried code base, we should try our best to write programs by using it. STL is a lot of important code, pre-developed for us. It is available as a library. We can write programs by using it. Thus our programs will be small and error free.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Lecture No. 43

Reading Material

Lecture 1, Lecture 25 - Lecture 42

Summary

- Programming Exercise - Matrices
- Design Recipe
- Problem Analysis
- Design Issues and Class Interface

vubookshoppk.com

Programming Exercise - Matrices

Mathematics is a good domain to develop different classes and programs. For example, solutions for Complex numbers, Matrices and Quadratic Equations can be sought for developing our own classes. In this lecture, we will take a problem to manipulate and perform different operations on Matrices. Matrices are used in lot of real world problems. We will perform problem analysis, design and implementation.

Let's take a look at analysis and design phases first by using our design recipe.

Design Recipe

Firstly we do analysis and try to come up with a problem statement. Express its essence, abstractly and with examples. After describing the problems in few sentences, we try to formulate the problem with examples. It is emphasized to pay attention to the details. We do analysis of the data structures to be used in the program and choose the best fit to the program requirements. The code is written to implement the program. After implementation is completed, we do its testing to verify that it is behaving properly in all scenarios. If any bugs are found, they are fixed. This cycle of testing and bug fixing continues until the program is working perfectly without any problem.
We are going to write a program to manage operations on Matrices.
At the start of the problem analysis phase, let's try to understand the problem domain first.

Problem Analysis

A matrix is nothing but a two-dimensional array of numbers. It is normally represented in rows and columns. A matrix is represented as:

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

It is a matrix *A* with 3 rows and 4 columns. So order of the matrix is 3 * 4.
Before going further, let's consider what are the operations normally performed on matrices.

- A matrix is added to another matrix.
- A scalar value (an ordinary number) is added to a matrix.
- A matrix is subtracted from another matrix.
- A scalar number is subtracted from a matrix.
- A matrix is multiplied with another matrix.
- A scalar number is multiplied with a matrix.
- A matrix is divided by a scalar.
- A matrix is transposed.

Now, we will define what these operations are and if there are any restrictions on matrices performing these operations.

The sum or addition of two matrices of the same order is found by adding the corresponding elements of the two matrices. If *A* and *B* are two matrices of order m * n to be added then their resultant matrix will also have the same order m * n.

$$A_{ij} + B_{ij}$$

Where i varies from 1 to m (max number of rows) and j varies from 1 to n (max number of cols).
Clearly, there is a restriction on the matrices performing this addition operation that they should have same numbers of rows and columns, in other words their order should be the same.
There is another operation of addition of scalar number to a matrix. In this operation, a number is added to all elements of the matrix.

Subtraction operation works in the same fashion that two matrices of the same order takes part in this operation and resultant matrix with similar order is obtained by subtracting each element of one matrix from the corresponding element of other matrix. For example, see the subtraction operation and assignment below:

$$C_{ij} = A_{ij} - B_{ij}$$

$$\begin{bmatrix} -2 & -4 & -5 \\ -2 & 2 & 0 \\ 0 & 0 & 10 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 5 & 6 & 7 \\ 9 & 10 & 11 \end{bmatrix} - \begin{bmatrix} 3 & 6 & 8 \\ 7 & 4 & 7 \\ 9 & 10 & 1 \end{bmatrix}$$

Not to confuse your understanding with assignment in computer programs, the resultant matrix is put on the left of assignment operator otherwise in Mathematics it is located on the right.
Each element of matrix *B* is subtracted from the corresponding element of the matrix *A* and the resultant goes to matrix C. *C* will have the same number of rows and columns as *A* and *B*.
Similar to the addition, there is another operation for subtracting a scalar from a matrix. In this case, a number is subtracted from each element of the matrix.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

For Division of a matrix by a scalar, the scalar number divides each element of the matrix. Let *x* be a scalar number and *A* be a matrix then division is represented as:

$$C_{ij} = A_{ij} / x$$

Each element of matrix *A* is divided by the number *x* to produce the corresponding number in the resultant matrix *C*. For example, A_{11} (element in first row and first column of matrix A) is divided by the scalar number *x* to provide C_{11} (element in first row and first column of matrix C).

The multiplication operation is bit complicated as compared to the above discussed operations. We will discuss simple case first, when a scalar is multiplied by a matrix. Suppose, this time we want to multiply the scalar *x* with the matrix *A* as:

$$C_{ij} = x * A_{ij}$$

Each element of matrix *A* is multiplied with the scalar *x* and the resultant number is put in the corresponding location inside the matrix C.

Now, we will see how a matrix is multiplied with another matrix. Firstly, there is a restriction on order of the matrices involved in this operation. The number of columns of the first matrix should be equal to the number of rows of the second matrix.
Two matrices are multiplied in the following manner:
We take the first row of first matrix and multiply it with the first column of the second matrix. The multiplication is done in such a way that the first element of the row is multiplied with the first element of the column, second element is multiplied with the second element and so on. The results of all these multiplication operations are added to produce one number. The resultant number is placed at the corresponding position (i.e. 1st row 1st col in this case) in the resultant matrix.
Further the same first row is multiplied with the second column of the second matrix and the resultant number is placed at intersecting position of first row and second column in the resultant matrix. This process goes on till the last column of the second matrix.
Then comes the second row of first matrix and whole operation is repeated for this row, this row is multiplied with all the columns of the second matrix. This process goes on till the last row of the first matrix.

$$\begin{bmatrix} (1)(2)+(2)(1) & (1)(4)+(2)(2) \\ (5)(2)+(6)(1) & (5)(4)+(6)(2) \end{bmatrix} \qquad \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} * \begin{bmatrix} 2 & 4 \\ 1 & 2 \end{bmatrix}$$

Note the resultant matrix is put on the left of the =. In Mathematics, this is put on right but not to confuse your understanding with assignment concept in computer programs, it is put on left.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

If a matrix with order *m* rows, *n* columns is multiplied with another matrix of *n* rows and *p* columns then the resultant matrix will have *m* rows and *p* columns. In the above diagram, the first matrix has two rows and second matrix has two columns, therefore, the resultant matrix has two rows and two columns.

Now comes the last operation, we are thinking of implementing i.e. Transpose of a matrix. Transpose of a matrix is obtained by interchanging its rows and columns. How do we interchange rows and columns for transposing the matrix? We take the first row of the matrix and write it as a first column of the new matrix. The second row of the original matrix is written as second column of the new matrix and similarly the last row of the original matrix is written as last column of the new matrix. At the end of this operation, when all rows of the original matrix are finished, we have new matrix as transpose of the original matrix. There is no change in the size (order or number of rows and cols of a matrix) of the transposed matrix when the original matrix is a square matrix. But when the original matrix is not a square matrix, there is a change in the order of the transposed matrix. The number of rows of the original matrix becomes the number of columns of the transposed matrix and the number of columns of the original matrix becomes the number of rows of the transposed matrix.

$$\begin{pmatrix} 1 & 2 & 3 \\ 5 & 6 & 7 \\ 9 & 10 & 11 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \end{pmatrix}$$

vubookshoppk.com

Until now in this problem analysis phase, we have analyzed the problem in order to understand what are the matrices and what are their operations to be implemented. Now at the next stage, we try to determine the followings:

- What are the constants to be used in our class?
- What are going to be the data structures to cater to the different sized matrices?
- How much memory is required and how it will be allocated?
- What is going to be the interface of the class?

Design Issues and Class Interface

We want to specify the size of the matrix at creation time and allocate the memory for that. So we don't see any use of constants inside our class named *Matrix*.

The size of the memory to be allocated is not going to be huge, as we are not catering to the very huge sized matrices. Therefore, the memory for a matrix is going to be allocated dynamically bluntly after the size of the matrix is specified in terms of rows and columns without worrying about the size of the matrix.

For the interface of our *Matrix* class, we will declare a constructor that will accept integer number of rows and columns of the matrix to be created as parameters.

```
Matrix ( int   rows, int   cols ) ;
```

The constructor function will be doing the memory allocation for the matrix.

As part of the interface, we will declare a display function inside our *Matrix* class that will display the elements on the screen.

void display (Matrix &);

To perform already discussed different operations on matrices, we need to overload operators. For example to perform addition of two matrices, + *operator* will be overloaded as a member function of the *Matrix* class. The + *operator* function will be called for the *Matrix* object on the left of the + and the *Matrix* object on the right to it will be passed as a parameter to it. This function will add the corresponding elements of the both matrices and returns the resultant back.

Matrix operator + (Matrix &) const;

The same thing applies to the subtraction operation of two matrices. – *operator* function will be overloaded for that as a member function of the Matrix class.

Matrix operator - (Matrix &) const;

The situation changes a bit, when we want to write the functions to cater to different operations where both the operands are not matrix objects rather one of them is scalar. For example, when we want to do the following operation:

A + x ;

Where A is a matrix and x is a scalar.
Then we write a member function that accepts a scalar number as a parameter instead of a Matrix object.

Matrix operator + (Scalar) const;

The Scalar can be an *int*, *double* or *float*, that we will cover later.

But the situation is more different, when we want to perform the scalar addition operation in the following manner:

x + A ;

By now we should be clear that member function cannot be written to handle this operation because there is a scalar number on the left of +. Therefore, we need to write a *friend operator* function for this type of operation. The friend functions are non-members and therefore, defined outside of the class.

friend Matrix operator + (Scalar , Matrix &) ;

Similarly, when a scalar is subtracted from a *Matrix* object like the following:

A - x ;

A member function is written to cater to this operation.

vubookshoppk.com

Matrix operator - (Scalar) const;

But again, when a matrix is subtracted from a scalar number:

x - A ;

Then we have to write a *friend operator* to handle this operation.

friend Matrix operator - (Scalar , Matrix &) ;

In order handle the multiplication operations of two Matrix objects like the following:

*A * B ;*

A member *operator* * function is defined.

*Matrix operator * (const Matrix &) ;*

This operator is called for the *Matrix* object on the left of * and the object on the right is passed as an argument. The function multiplies both the matrices and returns the resultant matrix.
When a scalar is multiplied with a scalar like:

*A * x ;*

The following member operator * handles this:

*Matrix operator * (Scalar) const;*

But for operation like the following:

*x * A;*

following friend *operator* function is written:

*friend Matrix operator * (const Scalar , const Matrix &) ;*

For division operation like the following:

A / x;

A member operator / is overloaded as:

Matrix operator / (const Scalar);

Now we will talk about transpose of a matrix. For this operation, we will write a member function *transpose* that will transpose the original matrix.

Matrix & transpose(void) ;

vubookshoppk.com

Now we are left with few more things to cover to complete the rudimentary interface of our class *Matrix*.

Operators += and -= are overloaded as member operators. These composite operators use the assignment operator (=).

We will also overload stream insertion and extraction operators as *friend* functions to our *Matrix* class as follows:

```
friend   ostream &   operator   <<  ( ostream & , Matrix & ) ;
friend   istream &   operator   >>  ( istream & , Matrix & ) ;
```

So here is how we declare our *Matrix* class. The interface of the class is the *public* methods of the class. Here is one important point to understand that what we are concerned about here is the class interface and not about the program interface to the user of the program. A programmer can develop user interface by writing his/her code while using the class interface.

```
/* Declaration of the Matrix class. This class is containing the double type elements */

class Matrix
{
   private:
      int numRows, numCols;
      double **elements;
   public:
      Matrix(int=0, int=0);         // default constructor
      Matrix(const Matrix & );      // copy constructor
       ~Matrix();                   // Destructor

       int getRows(void) const;      // Utility fn, returns no. of rows
       int getCols(void) const;      // Utility fn, returns no. of columns
       const Matrix & input(istream &is = cin);  // Read matrix from istream
       const Matrix & input(ifstream &is);       // Read matrix from istream
       void output(ofstream &os) const;          // Utility fn, prints matrix with graphics
       void output(ostream &os = cout) const;    // Utility fn, prints matrix with graphics

       const Matrix& transpose(void);     // Transpose the matrix and return a ref

       const Matrix & operator = (const Matrix &m);     // Assignment operator

       Matrix operator+( Matrix &m) const;        // Member op + for A+B; returns matrix
       Matrix operator + (double d) const;
       const Matrix & operator += (Matrix &m);
       friend Matrix operator + (double d, Matrix &m);

       Matrix operator-( Matrix & m) const;       // Member op + for A+B; returns matrix
       Matrix operator - (double d) const;
       const Matrix & operator -= (Matrix &m);
```

vubookshoppk.com

```
        friend Matrix operator - (double d, Matrix& m);

        Matrix operator*(const Matrix & m);
        Matrix operator * (double d) const;
        friend Matrix operator * (const double d, const Matrix& m);

        Matrix operator/(const double d);
        friend ostream & operator << ( ostream & , Matrix & );
        friend istream & operator >> ( istream & , Matrix & );
        friend ofstream & operator << ( ofstream & , Matrix & );
        friend ifstream & operator >> ( ifstream & , Matrix & );
       void   display( ) ;
};
```

In the above declarations, we should note how we are passing and returning *Matrix* objects. We are passing and returning the Matrix objects by reference because passing the Matrix objects by value will be a overhead that will affect performance and more memory will be allocated and de-allocated on stack.

Notice that we are doing dynamic memory allocation inside the *constructor* of the class. You must be remembering that wherever the dynamic memory allocation is made, it has to be freed explicitly. To de-allocate the memory, we will write code inside the *destructor* of the class *Matrix*. The other consideration when we are allocating memory on free store from within constructor is that the default *assignment operator* will not work here. Remember, the default *assignment operato*r makes *shallow copy* of the object members, therefore, we will have to write our own *assignment operator* (=) in order to make *deep copy* of the object data members. Remember that a *copy constructor* is called when a new *Matrix* object is initialized and constructed based on an already existent *Matrix* object. Therefore, we have to write our own *copy constructor* in order to make deep copy of the object data members.

There is one very important point to mention about this class *Matrix*. A *Matrix* can be composed of *int*s, *float*s or *double*s as their elements. Instead of handling these data types separately, we can write *Matrix* class as a template class and write code once for all native data types. While writing this template class, the better approach to write will be, to go with a simple data type (e.g. *double*) first to write a *Matrix* class and then extend it to a template class later. Another thing that can be templatized in the *Matrix* class is the *Scalar* number. Actually, this Scalar number can be an *int*, *float* or *double*; therefore, we may also use a template for this.

We have to perform certain checks and make decisions inside the implementation of member functions. For example, while writing the division operator member function, we will check against the number that it should be non-zero. Before adding two matrices, we will check for their number of rows and columns to be equal. Also in this exercise, we have declared only one class *Matrix* to manipulate matrices. There are alternate approaches to this. For example, we could declare a *Row* class first and then contain multiple objects (same in number as number of rows required for the matrix object) of *Row* class inside the *Matrix* class making a matrix of a certain size. To

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

make it simple, we have selected to manage matrices using only one class *Matrix*. The objective here is to practice the already studied programming constructs as much as possible.

vubookshoppk.com

Lecture No. 44

Reading Material

Lecture 25 - Lecture 43

Summary

- Matrix Class
- Definition of Matrix Constructor
- Destructor of Matrix Class
- Utility Functions of Matrix
  - Input Function
  - Transpose Function
- Code of the Program

vubookshoppk.com

Matrix Class

After talking at length about the concept of matrices in the previous lecture, we are going to have a review of 'code' today with special emphasis on concepts of constructors and destructors. We may also briefly discuss where should a programmer return by value, return by reference besides having a cursory look on the usage of the pointers.

The data structure of the *Matrix* class is very simple. We have defined an arbitrary number of functions and operators. You may add and subtract more functions in it. In this lecture, we have chosen just a few as it is not possible to discuss each and every one in a brief discourse. As discussed earlier, the code is available to use that can be compiled and run. This class is not complete and a lot of things can be added to it. You should try to enhance it, try to improve and add to its functionality.

One of the things that a programmer will prefer to do with this class is its templatization. We have implemented it for type *double*. It was done due to the fact that the elements of the *Matrix* are of type *double*. You may like to improve and make it a templatized class so that it can also handle integer elements. This class is written in a very straightforward manner. The *double* is used only for elements to develop it into a *Matrix* class for integers if you replace all the *double* word with the *int*. Be careful, you cannot revert it back to *double* by just changing all the *int* to *double* as integers are used other than element types.

Let's discuss the code beginning with the data structure of the class. In keeping the concepts of data hiding and encapsulation, we have put the data in the private section of the class. We have defined number of rows (i.e. *numRows*) and number of columns

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

(i.e. *numCols*) as integers. These will always be whole number. As we cannot have one and a half row or column, so these are integers.

The private part of the *Matrix* class is:

```
int numRows, numCols;
double **elements;
```

These are fixed and defined. So whenever you have an object of this class, it will have a certain number of rows and certain number of columns. These are stored in the variables- *numRows* and *numCols*. Where will be the values of the elements of this matrix? The next line is *double **elements;* i.e. *elements* is an array of pointers to *double*. First * is for array and second * makes it a pointer. It means that '*elements'* is pointing to a two-dimension array of *double*. When we say *elements[i]*, it means that it is pointing to an array of *double*. If we say *elements[i][j]*, it means we are talking about a particular element of type *double*. We have not taken the two-dimension array in the usual way but going to dynamic memory allocation. We have developed a general *Matrix* class. The objects created from it i.e. the instances of this class, could be small matrices as 2*2. These may also be as big matrix as 20*20. In other words, the size of the matrix is variable. In fact, there is no requirement that size should be square. It may not be 20*20. It may be a matrix of 3*10 i.e. three rows and ten columns. So we have complete flexibility. When we create an object, it will store the number of rows in *numRows* and number of columns in *numCols*. The elements will be dynamically allocated memory in which the *double* value is stored.

While using the dynamic memory, it is good to keep in mind that certain things are necessary to be implemented in the class. 1) Its constructor should make memory allocation. We will use the *new* operator, necessitating the need of defining its destructor also. Otherwise, whenever we create an object, the memory will be allocated from the free store and not de-allocated resulting in the wastage of the memory. Therefore a destructor is necessary while de-allocating memory. 2) The other thing while dealing with classes having dynamic memory allocation, we need to define an assignment operator. If we do not define the assignment operator, the default will do the member wise copy i.e. the shallow copy. The value of pointer will be copied to the pointer but the complete data will not be copied. We will see in the code how can we overcome this problem..

Let's discuss the code in the public interface of the class. The first portion of the public interface is as:

```
Matrix(int=0, int=0);        // default constructor
Matrix(const Matrix & );   // copy constructor
~Matrix();                   // Destructor
```

In the public interface, the first thing we have is the constructors. In the default constructor, you see the number of columns and number of rows. The default values are zero. If you just declare a matrix as *Matrix m*, it will be an object of class *Matrix* having zero rows and zero columns i.e. memory is not allocated yet. Then we have also written a copy constructor. Copy constructor becomes necessary while dealing

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

with dynamic memory allocation in the class. So we have to provide constructor, destructor, assignment operator and the copy constructor. The declaration line of copy constructor is always the same as the name of the class. In the argument list, we have a reference to an object of the same class i.e. *Matrix*(const *Matrix* &); The & represents the reference. This is the prototype. After constructors, we have defined a destructor. Its prototype is also standard. Here it is ~*Matrix*(); it takes no argument and returns nothing. Remember that constructors and destructors return nothing.

After this, we have utility functions for the manipulation of the *Matrix*.

```
int getRows(void) const;                    // Utility fn, returns no. of rows
int getCols(void) const;                    // Utility fn, returns no. of columns

const Matrix & input(istream &is = cin);  // Read from istream i.e. keyboard
const Matrix & input(ifstream &is);         // Read matrix from ifstream

void output(ofstream &os) const;            // Utility fn, prints matrix with graphics
void output(ostream &os = cout) const;  // Utility fn, prints matrix with graphics

const Matrix& transpose(void);            // Transpose the matrix and return a ref
```

We have defined two small functions as *getRows* and *getCols* which will return the number of rows and number of columns of the matrix respectively. Here, you are writing the class and not the client which will use this class. During the usage of this class, there may be need of some more functions. You may need to add some more functionality depending on its usage. At this point, a function may be written which will return some particular row as a vector or the nth column of a matrix. These things are left for you to do.

We need some function to input values into the matrix. There are two input functions both named as *input*. One is used to get the value from the keyboard while the other to get the values from some file. There is a little bit difference between the declaration of these two that will be discussed later. The *input* function which will get the input from the keyboard, takes an argument of type *istream*. Remember that *cin,* that is associated with the keyboard, is of type *istream*. These are member functions and called by some object of *Matrix*. The *Matrix* object will be available to it through *this* pointer. The *istream* is passed as argument and we have given a temporary name to the input argument i.e. *is* and its default value is *cin*. It is a nice way of handling default values. If you write in the main program as *m.input()* where *m* is an object of type *Matrix*, it will get the input from the keyboard. This is due to the fact that it is getting *cin* by default. We have defined another *input* function and that is an example of function overloading. This function takes *ifstream* as input argument i.e. a file stream. If we want to read the matrix data from some file, this *input* function may be employed. There is no default argument in it.

Similarly we have two *output* functions. The names are chosen as arbitrary. You may want to use print or display. One output function will display the matrix on the screen. Its argument is *ostream* while the default value will be *cout*. The other output function will be used when we want to write the matrix in some file. It takes *ofstream* as

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

argument and we have not provided any default argument to it. You will have to provide a file handle to use this function.

Let's continue to talk about the arithmetic manipulations we want to do with the matrices.

```
Matrix operator+( Matrix &m) const;   // Member op + for A+B; returns matrix
Matrix operator + (double d) const;      // Member op + for A+d; returns matrix
const Matrix & operator += (Matrix &m); // Member op += for A +=B
friend Matrix operator + (double d, Matrix &m); // friend operator for d+A

Matrix operator-( Matrix & m) const;   // Member op - for A-B; returns matrix
Matrix operator - (double d) const;       // Member op - for A-d;
const Matrix & operator -= (Matrix &m);  // Member op -= for A-=B;
friend Matrix operator - (double d, Matrix& m); // Friend op - for d-A;

Matrix operator*(const Matrix & m); // Member op * for A*B;
Matrix operator * (double d) const; // Member op * for A*d;
friend Matrix operator * (const double d, const Matrix& m);//friend op*,  d*A

const Matrix& transpose(void);     // Transpose the matrix and return a ref
const Matrix & operator = (const Matrix &m);     // Assignment operator
```

We have defined different functions for plus. Some of these are member operators while the others called as friend. The first plus operator is to add two matrices. It is a member operator that takes a *Matrix* object as argument. The second one is to add some *double* number to matrix. Remember that we are having a class of *Matrix* with *double* elements. So we will add *double* number to it. It is also a member operator. When you write something like *m + d*, where *m* is an object of type *Matrix* and *d* is a *double* variable, this operator will be called. Here *Matrix* object is coming on the left-hand side and will be available inside the operator definition by this pointer. On the other hand, the *double* value *d* is presented as argument to the operator.

There is another variety of adding double to the matrix i.e. if we write as *d + m*. On the left-hand side, we don't have *Matrix*. It cannot be a member function as the driving force is not an object of our desired class. If not a member function, it will have two arguments. First argument will be of type *double* while the second one is going to be of *Matrix* object in which we will add the *double* number. As it is not a member function and we want to manipulate the private data members of the class, it has to be a friend function. Function will be defined outside, at file level scope. In other words, it will not be a member function of class *Matrix* but declared here as a friend. It is defined as:

```
Friend Matrix operator + (double d, Matrix &m);
```

Its return type is *Matrix*. When we add a *double* number, it will remain as *Matrix* .We will be able to return it. The final variant is included as an example of code reuse i.e. the += operator. We write in our program *i += 3;* and it means *i = i + 3*; It would be nice if we can write *A += B* where *A* and *B* both are matrices.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

After plus, we can do the same thing with the minus operator. Having two matrices-*A* and *B*, we want to do *A-B*. On the left hand side, we have *Matrix*, so it will be a member operator. The other matrix will be passed as argument. In *A-B, A* is calling this operator while *B* being passed as argument. We can also do *A-d* where *A* is a *Matrix* and *d* is of type double. For this, we will have to write a member operator. All these operators are overloaded and capable of returning *Matrix*. In this overloaded operator, *double* will be passed as argument. Then we might want to do it as *d - A*, where *d* is *double* variable and *A* is of type *Matrix*. Since the left hand side of the minus (-) is *double*, we will need a friend function, as it is not possible to employ a member function. Its prototype is as:

```
friend Matrix operator - (double d, Matrix& m);
```

Let's now talk about multiplication. We have discussed somewhat about it in the previous lecture. For multiplication, the first thing one needs to do is the multiplication of two matrices i.e. *A*B* where *A* and *B* both are matrices. As on the left hand side of the operator * we have a *Matrix* so it will be a member function so *A* can be accessed through *this* pointer. *B* will be passed as argument. A matrix should be returned. Thus, we have a member operator that takes an argument of type *Matrix* and returns a *Matrix*. You may like to do the same thing, which we did with the plus and minus. In other words, a programmer will multiply a matrix with a *double* or multiply a *double* with a matrix. In either case, we want that a matrix should be returned. So at first, *A * d* should be a member function. Whereas *d * A* will be a friend function. Again the return type will be a *Matrix*.

In case of division, we have only one case i.e. the division of the matrix with a *double* number. This is *A / d* where *A* is a *Matrix* while *d* is a *double* variable. We will divide all the elements of the matrix with this *double* number. This will return a matrix of the same size as of original.

Taking benefit of the stream insertion and extraction operator, we can use double greater than sign (>>) and write as >> *m* where *m* is a *Matrix*. For this purpose, we have written stream extraction operator. The insertion and extraction functions will be friend functions as on the left-hand side, there will be either input stream or output stream.

We have also defined assignment function in it. As we are using dynamic memory allocation in the class, assignment is an important operator. Another important function is transpose, in which we will interchange the rows into columns and return a *Matrix*. This is the interface of our *Matrix* class. You may want to add other functions and operators like +=, -=, *=. But it is not possible to add /= due to its very limited scope.

We have used the keyword const in our class. You will find somewhere the return type as *Matrix* and somewhere as *Matrix &*. We will discuss each of these while dealing with the code in detail.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Definition of Matrix Constructor

Let's start with the default constructor. Its prototype is as under:

```
Matrix(int = 0, int = 0);        // default constructor
```

We are using the default argument values here. In the definition of this function, we will not repeat the default values. Default values are given at one place. The definition code is as:

```
Matrix::Matrix(int row, int col)        //default constructor
{
   numRows = row;
   numCols = col;

   elements = new (double *) [numRows];

   for (int  i = 0; i < numRows; i++){
      elements[i] = new double [ numCols];

      for(int j = 0; j < numCols; j++)
            elements[i][j] =0.0; // Initialize to zero
   }
}
```

vubookshoppk.com

Two integers are passed to it. One represents the number of rows while the other is related to the number of columns of the *Matrix* object that we want to create. At first, we will assign these values to *numRows* and *numCols* that form a part of the data structure of our class. Look at the next line. We have declared *elements* as ***elements* i.e. the array of pointers to *double*. We can directly allocate it by getting the elements of *numRows* times *numCols* of type *double* from free store. But we don't have the *double* pointer. Therefore, first of all, we say that element is array of pointer to *double* and allocate pointers as much as needed. So we use the *new* operator and use *double** cast. It means that whatever is returned is of type pointer to *double*. How many pointers we need? This is equal to number of rows in the matrix. There is one pointer for each of the rows i.e. now a row can be an array. This is a favorable condition, as we have to enter data in the columns. Now *elements* is *numRows* number of pointers to *double*. After having this allocation we will run a loop.

In C/C++ languages, every row starts from zero and ends with *upper-limit – 1*. Now in the loop, we have to allocate the space for every *elements[i];* How much space is needed here? This will be of type *double* and equal to number of columns. So when we say *new double[numCols],* it means an array of *double*. As *elements[i]* represents a pointer and now it is pointing to an array. Remember that pointers and arrays are synonymous. Now a pointer is pointing to this array. We have the space now and want to initialize the matrix. For this, we have written another loop. To assign the value, we will write as:

```
elements[i][j] = 0.0;
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Let's review this again. First of all we have assigned the values to *numRows* and *numCols*. Later, we allocated the pointers of *double* from the free store and assigned to *elements*. Then we got the space for each of this pointer to store *double*. Finally, we initialized this space with 0.0. Now we have a constructive matrix.

Is there any exceptional value? We can think of assigning negative values to number of rows or number of columns. If you want to make sure that this does not happen, you can put a test by saying that *numRows* and *numCols* must be greater than or equal to zero. Passing zero is not a problem as we have zero space and nothing happens actually. Now we get an empty matrix of dimension 0*0. But any positive number supplied will give us a constructor and initialize zero matrix.

Let's discuss the other constructor. This is more important in the view of this class especially at a time when we are going to do dynamic memory allocation. This is copy constructor. It is used when we write in our program as *Matrix A(B);* where *A* and *B* both are matrices. We are going to construct *A* while *B* already exists. Here we are saying that give us a new object called *A* which should be identical to the already existing object *B*. So it is a copy constructor. We are constructing an object as a copy of another one that already exists. The other usage of this copy constructor is writing in the main function as *Matrix A = B;* Remember that this is declaration and not assignment statement. Here again copy constructor will be called. We are saying that give us a duplicate of *B* and its name should be *A*. Here is the code of this function:

```
Matrix::Matrix(const Matrix &m)
{
   numRows = m.numRows;
   numCols = m.numCols;
   elements = new (double *) [numRows];
   for (int  i = 0; i < numRows; i++){
      elements[i] = new double [ numCols];
      for(int j = 0; j < numCols; j++)
            elements[i][j] = m.elements[i][j];
   }
}
```

We are passing it a constant reference of *Matrix* object to ensure that the matrix to be copied is not being changed. Therefore we make it *const*. We are going to create a brand new object that presently does not exist. We need to repeat the code of regular constructor except the initialization part. Its rows will be equal to the rows of the object supplied i.e. *numRows = m.numRows*. Similarly the columns i.e. *numCols= m.numCols*. Now we have to allocate space while using the same technique earlier employed in case of the regular constructor. In the default constructor, we initialize the elements with zero. Here we will not initialize it with zero and assign it the value of *Matrix m* as *elements[i][j] = m.elements[i][j]*. Remember that we use the dot operator to access the data members. We have not used the dot operator on the left hand side. This is due to the fact that this object is being constructed and available in this function through *this* pointer. Therefore the dot operator on the left hand side is not needed. We will use it on the right hand side to access the data members of *Matrix m*. This is our copy constructor. In this function, we have taken the number of rows and columns of the object whose copy is being made. Then we allocate it the space

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

and copy the values of elements one by one. The other thing that you might want to know is the use of nested loop both in regular constructor and the copy constructor.

Destructor of Matrix Class

'Destructor' is relatively simple. It becomes necessary after the use of new in the constructor. While creating objects, a programmer gets memory from the free store. So in the destructor, we have to return it. We will do it as:

```
delete [] elements;
```

Remember that '*elements*' is the variable where the memory has been allocated. The [] simply, indicates that it is an array. The compiler automatically takes care of the size of the array and the memory allocated after being returned, goes back to the free store. There is only one line in the destructor. It is very simple but necessary in this case.

Utility Functions of Matrix

The functions *getRows()* and *getCols()* are relatively simple. They do not change anything in the object but only read from the object. Therefore we have made this function constant by writing the *const* keyword in the end. It means that it does not change anything. The code of the *getRows()* functions is as follows:

```
int Matrix :: getRows ( ) const
{
   return numRows;
}
```

This function returns an *int* representing the number of rows. It will be used in the main function as *i = m.getRows();* where *i* is an *int* and *m* is a *Matrix* object. Same thing applies to the *getCols()* function. It is of type *const* and returns an *int* representing the number of columns.

Let's talk about little bit more complicated function. It is the output to the screen functions. We want that our matrix should be displayed on the screen in a beautiful way. You have seen that in the books that matrix is written in big square brackets. The code of the function is as:

```
void Matrix::output(ostream &os) const
{
   // Print first row with special characters
   os.setf(ios::showpoint);
   os.setf(ios::fixed,ios::floatfield);
   os << (char) 218;

   for(int j = 0; j < numCols; j++)
         os << setw(10) << " ";

   os << (char) 191 << "\n";
```

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

```
    // Print remaining rows with vertical bars only
    for (int i = 0; i < numRows; i++){
        os << (char) 179;
        for(int j = 0; j < numCols; j++)
              os << setw(10)<< setprecision(2) << elements[i][j];
        os << (char) 179 << "\n";
    }

     // Print last row with special characters
     os << (char) 192;
     for(int j = 0; j < numCols; j++)
           os << setw(10) << " ";

     os << (char) 217 << "\n";
}
```

We have used special characters that can be viewed in the command window. We have given you an exercise of printing the ASCII characters on the screen. After ASCII code 128, we have special graphic symbols. We have used the values of those symbols here. To print those in the symbol form, we have forced it to be printed as *char*. If we do not use the *char*, it would have printed the number 218 i.e. it would have written an integer. We have forced it to print the character whose ASCII value is 218. Now it prints the graphic symbol for that character. We have referenced the ASCII table and seen which symbol will fit in the left corner i.e. 218. So we have written it as:

```
os << (char) 218;
```

os is the output stream. *char* is forcing it to be print as character. The left corner will be printed on the screen as ⌈. Now we need the space to print the columns of the matrix. In the first line we will print the spaces using a loop as:

```
for(int j = 0; j < numCols; j++)
  os << setw(10) << " ";
```

Here we have changed the width as 10. You can change it to whatever you like. Then we print nothing in the space of ten characters and repeat that for the number of columns in the matrix. After this, we printed the right corner. This is the first line of the display. Other lines will also contain the values of the elements of the matrix. These lines will start with a vertical line and then the element values of the row and in the end we have a vertical line. For each row, we have a vertical bar and the number values which will be equal to number of columns (elements in each row equals to the number of columns) and then a vertical bar in the end. In the beginning of this code, we have used two other utilities to improve the formatting. Here we have a matrix of type *double* so every element of the matrix is *double*. For *double*, we have used *os.setf(ios::fixed,ios::floatfield);* that means that it is a fixed display. Scientific notation will not be used while decimal number is displayed with the decimal point. After this, we have set the precision with two number of places. So we have a format and our decimal numbers will always be printed with two decimal places. The numbers are being printed with a width of ten characters so the last three places will

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

be as *.xx*. Rest of the code is simple enough. We have used the nested loops. Whenever you have to use rows and columns, it will be good to use nested loops. When all the rows have been printed, we will print the below corners. We referenced the ASCII table, got the graphic symbol, printed it, left the enough space and then printed the other corner. The matrix is now complete. When this is displayed on the screen, it seems nicely formatted matrix with graphic symbols. That is our basic output function.

Let's look at the file output function. While doing the output on the screen, we made it nicely formatted. Now you may like to store the matrix in a file. While storing the matrix in the file, there is no need of these lines and graphic symbol. We only need its values to read the matrix from the file. So there is a pair of functions i.e. output the matrix in the file and input from the file. To write the output function, we actually have to think about the input function.

Suppose, we have declared a 2*2 *Matrix m* in our program. Somewhere in the program, we want to populate this matrix from the file. Do we know that we have a 2*2 matrix in the file. How do we know that? It may 5*5 or 7*3 matrix. So what we need to do is somehow save the number of rows and columns in the file as well. So the output function that puts out on the file must put out the number of rows and number of columns and then all of the elements of the matrix. Following is the code of this function:

vubookshoppk.com

```
void Matrix::output(ofstream &os) const
{
    os.setf(ios::showpoint);
    os.setf(ios::fixed,ios::floatfield);

    os << numRows << " " << numCols << "\n";

    for (int i = 0; i < numRows; i++){

        for(int j = 0; j < numCols; j++)
            os << setw(6) << setprecision(2) << elements[i][j];

        os << "\n";
    }
}
```

The code is shorter than the other output function due to non-use of the graphical symbols. First of all, we output the number of rows and number of columns. Then for these rows and columns, data elements are written. We have also carried out a little bit formatting. While seeing this file in the notepad, you will notice that there is an extra line on the top that depicts the number of rows and columns.

Input Functions

Input functions are also of two types like output functions. The first function takes input from the keyboard while the other takes input from the file. The function that takes input from the keyboard is written in a polite manner because humans are interacting with it. We will display at the screen "Input Matrix size: 3 rows by 3

columns" and it will ask "Please enter 3 values separated by spaces for row no. 1" for each row. Spaces are delimiter in C++. So spaces will behave as pressing enter from the keyboard. If you have to enter four numbers in a row, you will enter as number (space) number (space) number (space) number before pressing the enter key. We have a loop inside which will process input stream and storing these values into *elements[i][j];* So the difference between this function and the file input function is 1) It prompts to the user and is polite. 2) It will read from the keyboard and consider spaces as delimiter.

The other input function reads from the file. We have also stored the number of rows and number of columns in the file. The code of this function is:

```
const Matrix & Matrix::input(ifstream &is)
{
    int Rows, Cols;
    is >> Rows;
    is >> Cols;
    if(Rows > 0 && Cols > 0){
        Matrix temp(Rows, Cols);
        *this = temp;
        for(int i = 0; i < numRows; i++){
            for(int j = 0; j < numCols; j++){
                is >> elements[i][j];
            }
        }
    }
    return *this;
}
```

First of all, we will read the number of rows and number of columns from the file. We have put some intelligence in it. It is better to check whether *numRows* and *numCols* is greater than zero. If it is so, then do something. Otherwise, there is nothing to do. If rows and columns are greater than zero, then there will be a temporary matrix specifying its rows and columns. These values are read from the file, showing that we have a matrix of correct size. Now this matrix is already initialized to zero by our default constructor. We can do two things. We can either read the matrix, return the value or we can first assign it to the matrix that was calling this function. We have assigned it first as **this = temp;* here *temp* is a temporary matrix which is created in this function but **this* is whatever this points to. Remember that this is a member function so *this* pointer points to the matrix that is calling this function. All we have to do is to assign the *temp* to the matrix, which is calling this function. This equal to sign is our assignment operator, which we have defined in our *Matrix* class. If the dimensions of the calling matrix are not equal to the *temp* matrix, the assignment operator will correct the dimensions of the calling matrix. It will assign the values, which in this case is zero so far. Now we will read the values from the file using the nested loops. The other way is to read the values from the file and populate the *temp* matrix before assigning it to the calling matrix in the end. That is the end of the function. Remember that the *temp* matrix, which we have declared in this function, will be destroyed after the exit from the function. This shows that the assignment operator is important here. All the values will be copied and it will perform a deep

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

copy. Does this function return something? Its return type is reference to a *const Matrix*. Its ending line is return **this* that means return whatever *this* points to and it is returned as reference. The rule of thumb is whenever we are returning the *this* pointer, it will be returned as a reference because this is the same object which is calling it. When you are returning a matrix that is not a reference, it is a returned by value. The complete matrix will be copied on the stack and returned. This is slightly wasteful. Yet you cannot return a reference to the *temp* object in this code. The reference of the *temp* will be returned but destroyed when the function is finished. The reference will be pointing to nothing. So you have to be careful while returning a reference to *this*.

Transpose Function

The transpose of a matrix will interchange rows into columns. There are two alternative requirements. In the first case, we have a square matrix i.e. the number of rows is equal to number of columns. In this situation, we don't need extra storage to do this. If the number of rows is not equal to the number of columns, then we have to deal it in a different way. We can use general case for both purposes but you will notice that it is slightly insufficient. Here is the code of the function.

```
const Matrix & Matrix::transpose()
{
    if(numRows == numCols){    // Square matrix
       double temp;
       for(int i = 0;i < numRows; i++){
            for(int j = i+1; j < numCols; j++){
                 temp = elements[i][j];
                 elements[i][j] = elements[j][i];
                 elements[j][i] = temp;
            }
       }
    }
    else  // not a square matrix
    {
       Matrix temp(numCols, numRows);
       for(int i = 0; i < numRows; i++){
            for(int j = 0; j < numCols; j++){
                 temp.elements[j][i] = elements[i][j];
            }
       }
       *this = temp;
    }
    return *this;
}
```

In the beginning, we checked the case of square matrix i.e. if the number of rows is equal to number of columns. Here we are dealing with the square matrix. We have to change the rows into columns. For this purpose, we need a temporary variable. In this case, it is a variable of type *double* because we are talking about a *double* matrix. Look at the loop conditions carefully. The outer loop runs for *i = 0* to *i < numRows*

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

and the inner loop runs from *j = i+1* to *j < numCols*. Then we have standard swap functionality. We have processed one triangle of the matrix. If you start the inner loop from zero, think logically what will happen. You will interchange a number again and again, but nothing will happen in the end, leaving no change in the matrix. This is the case of the square matrix. But in case of non-square matrix i.e. the code in the else part, we have to define a new matrix. Its rows will be equal to the columns of the calling matrix and its columns will be equal to the number of rows. So we have defined a new *Matrix temp* with the number of rows and columns interchanged as compared to the calling matrix. Its code is straightforward. We are doing the element to element copy. The difference is, in the loop we are placing the *x row, y col* element of the calling matrix to *y row, x col* of the *temp* matrix. It is an interchange of the rows and columns according to the definition of the transpose. When we have all the values copied in the *temp*. We do our little magic that is **this = temp*. Which means whatever *this* points to, now assigned the values of the matrix *temp*. Now our horizontal matrix becomes vertical and vice versa. In the end, we return *this*. This is the basic essence of transpose code.

We will continue the discussion on the code in the next lecture. We will look at the assignment operator, stream operator and try to recap the complete course.

Code of the Program

The complete code of the matrix class is:

vubookshoppk.com

```
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <stdio.h>
#include <fstream.h>

class Matrix
{
   private:
      int numRows, numCols;
      double **elements;
   public:
      Matrix(int=0, int=0);        // default constructor
      Matrix(const Matrix & );     // copy constructor
        ~Matrix();                 // Destructor

        int getRows(void) const;     // Utility fn, returns no. of rows
        int getCols(void) const;     // Utility fn, returns no. of columns
        const Matrix & input(istream &is = cin);  // Read matrix from istream
        const Matrix & input(ifstream &is);       // Read matrix from istream
        void output(ofstream &os) const;          // Utility fn, prints matrix with graphics
        void output(ostream &os = cout) const;    // Utility fn, prints matrix with graphics

        const Matrix& transpose(void);     // Transpose the matrix and return a ref

        const Matrix & operator = (const Matrix &m);     // Assignment operator
```

```
        Matrix operator+( Matrix &m) const;        // Member op + for A+B; returns matrix
        Matrix operator + (double d) const;
        const Matrix & operator += (Matrix &m);
        friend Matrix operator + (double d, Matrix &m);

        Matrix operator-( Matrix & m) const;       // Member op + for A+B; returns matrix
        Matrix operator - (double d) const;
        const Matrix & operator -= (Matrix &m);
        friend Matrix operator - (double d, Matrix& m);

        Matrix operator*(const Matrix & m);
        Matrix operator * (double d) const;
        friend Matrix operator * (const double d, const Matrix& m);

        Matrix operator/(const double d);
        friend ostream & operator << ( ostream & , Matrix & );
        friend istream & operator >> ( istream & , Matrix & );
        friend ofstream & operator << ( ofstream & , Matrix & );
        friend ifstream & operator >> ( ifstream & , Matrix & );


};
Matrix::Matrix(int row, int col)       //default constructor
{
   numRows = row;
   numCols = col;
   elements = new (double *) [numRows];
   for (int  i = 0; i < numRows; i++){
      elements[i] = new double [ numCols];
      for(int j = 0; j < numCols; j++)
            elements[i][j] = 0; // Initialize to zero
   }
}

Matrix::Matrix(const Matrix &m)
{
   numRows = m.numRows;
   numCols = m.numCols;
   elements = new (double *) [numRows];
   for (int  i = 0; i < numRows; i++){
      elements[i] = new double [ numCols];
      for(int j = 0; j < numCols; j++)
             elements[i][j] = m.elements[i][j];
   }
}

Matrix::~Matrix(void)
{
    delete [] elements;
```

vubookshoppk.com

```
}

int Matrix :: getRows ( ) const
{
 return numRows;
}

int Matrix :: getCols ( ) const
{
 return numCols;
}
void Matrix::output(ostream &os) const
{
    // Print first row with special characters
   os.setf(ios::showpoint);
   os.setf(ios::fixed,ios::floatfield);
    os << (char) 218;
    for(int j=0; j<numCols; j++)
          os << setw(10) << " ";
    os << (char) 191 << "\n";
    // Print remaining rows with vertical bars only
    for (int i=0; i<numRows; i++){
       os << (char) 179;
       for(int j=0; j<numCols; j++)
             os << setw(10)<< setprecision(2) << elements[i][j];
       os << (char) 179 << "\n";
     }
     // Print last row with special characters
    os << (char) 192;
    for(int j=0; j<numCols; j++)
          os << setw(10) << " ";
    os << (char) 217 << "\n";

}

void Matrix::output(ofstream &os) const
{
          os.setf(ios::showpoint);
    os.setf(ios::fixed,ios::floatfield);
    os << numRows << " " << numCols << "\n";
    for (int i=0; i<numRows; i++){
       for(int j=0; j<numCols; j++)
             os << setw(6) << setprecision(2) << elements[i][j];
       os << "\n";
     }

}

const Matrix & Matrix::input(istream &is)
{
```

vubookshoppk.com

```
    cout << "Input Matrix size: " << numRows << " rows by " << numCols << " columns\n";
    for(int i=0; i<numRows; i++){
        cout << "Please enter " << numCols << " values separated by spaces for row no." << i+1 << ": ";
        for(int j=0; j<numCols; j++){
            cin >> elements[i][j];
        }
    }
    return *this;
}

const Matrix & Matrix::input(ifstream &is)
{
    int Rows, Cols;
    is >> Rows;
    is >> Cols;
    if(Rows>0 && Cols > 0){
        Matrix temp(Rows, Cols);
        *this = temp;
        for(int i=0; i<numRows; i++){
            for(int j=0; j<numCols; j++){
                is >> elements[i][j];
            }
        }
    }
    return *this;
}

const Matrix & Matrix::transpose()
{
    if(numRows == numCols){    // Square matrix
        double temp;
        for(int i=0; i<numRows; i++){
            for(int j=i+1; j<numCols; j++){
                temp = elements[i][j];
                elements[i][j] = elements[j][i];
                elements[j][i] = temp;
            }
        }
    }
    else
    {
        Matrix temp(numCols, numRows);
        for(int i=0; i<numRows; i++){
            for(int j=0; j<numCols; j++){
                temp.elements[j][i] = elements[i][j];
            }
        }
        *this = temp;
```

vubookshoppk.com

```
      }
      return *this;
}

const Matrix & Matrix :: operator = ( const Matrix & m )
{
 if( &m != this){
      if (numRows != m.numRows || numCols != m.numCols){
        delete [] elements;
        elements = new (double *) [m.numRows];
        for (int  i = 0; i < m.numRows; i++)
           elements[i]=new double[m.numCols ];
      }
      numRows = m.numRows;
      numCols = m.numCols;
      for ( int i=0; i<numRows; i++){
         for(int j=0; j<numCols; j++){
               elements[i][j] = m.elements[i][j];
         }
      }
  }
  return *this;
}

Matrix Matrix::operator + (  Matrix &m ) const
{
 // Check for conformability
 if(numRows == m.numRows && numCols == m.numCols){
   Matrix temp(m);
   for (int  i = 0; i < numRows; i++){
     for (int j = 0; j < numCols; j++){
         temp.elements[i][j] +=  elements[i][j];
     }
   }
   return temp ;
  }
}

Matrix Matrix::operator + (  double d ) const
{
   Matrix temp(*this);
   for (int  i = 0; i < numRows; i++){
     for (int j = 0; j < numCols; j++){
         temp.elements[i][j] +=  d;
     }
   }
  return temp ;
}

const Matrix & Matrix::operator += (Matrix &m)
```

vubookshoppk.com

```
{
      *this = *this + m;
      return *this;
}

Matrix Matrix::operator - (  Matrix &m ) const
{
 // Check for conformability
 if(numRows == m.numRows && numCols == m.numCols){
   Matrix temp(*this);
   for (int  i = 0; i < numRows; i++){
     for (int j = 0; j < numCols; j++){
         temp.elements[i][j] -=  m.elements[i][j];
     }
   }
   return temp ;
  }
}

Matrix Matrix::operator - (  double d ) const
{
   Matrix temp(*this);
   for (int  i = 0; i < numRows; i++){
     for (int j = 0; j < numCols; j++){
         temp.elements[i][j] -=  d;
     }
   }
  return temp ;
}

const Matrix & Matrix::operator -= (Matrix &m)
{
      *this = *this - m;
      return *this;
}

Matrix Matrix::operator* ( const Matrix& m)
{
   Matrix temp(numRows,m.numCols);
   if(numCols == m.numRows){
      for ( int i = 0; i < numRows; i++){
         for ( int j = 0; j < m.numCols; j++){
            temp.elements[i][j] = 0.0;
            for( int k = 0; k < numCols; k++){
                temp.elements[i][j] += elements[i][k] * m.elements[k][j];
            }
         }
     }
   }
   return temp;
```

vubookshoppk.com

```
}

Matrix Matrix :: operator * ( double d) const
{
   Matrix temp(*this);
   for ( int i = 0; i < numRows; i++){
     for (int  j = 0; j < numCols; j++){
        temp.elements[i][j] *= d;
     }
   }
   return temp;
}

Matrix operator * (const double d, const Matrix& m)
{
   Matrix temp(m);
   temp = temp * d;
   return temp;
}

Matrix Matrix::operator / (const double d)
{
      Matrix temp(*this);
      for(int i=0; i< numRows; i++){
            for(int j=0; j<numCols; j++){
                  temp.elements[i][j] /= d;
            }
      }
      return temp;
}

Matrix operator + (double d, Matrix &m)
{
      Matrix temp(m);
      for(int i=0; i< temp.numRows; i++){
            for(int j=0; j<temp.numCols; j++){
                  temp.elements[i][j] *= d;
            }
      }
      return temp;
}

Matrix operator - (double d, Matrix& m)
{
      Matrix temp(m);
      for(int i=0; i< temp.numRows; i++){
            for(int j=0; j<temp.numCols; j++){
                  temp.elements[i][j] = d - temp.elements[i][j];
            }
      }
```

vubookshoppk.com

```
    return temp;
}

ostream & operator  << ( ostream & os, Matrix & m)
{
    m.output();
    return os;
}

istream & operator  >> ( istream & is, Matrix & m)
{
    m.input(is);
    return is;
}

ofstream & operator  << ( ofstream & os, Matrix & m)
{
    m.output(os);
    return os;
}

ifstream & operator  >> ( ifstream & is, Matrix & m)
{
    m.input(is);
    return is;
}

int main()
{
  // declaring two matrices
  Matrix m(4,5), n(5,4);

  // getting input from keyboard
  cout << "Taking the input for m(4,5) and n(5,4) \n";
  m.input();
  n.input();

  // displaying m and taking its transpose
  cout << "Displaying the matrix m(4,5) and n(5,4)\n";
  m.output();
  n.output();

  cout << "Taking the transpose of matrix m(4,5) \n";
  m.transpose();

  cout << "Displaying the matrix m(5,4) and n(5,4) \n";
  m.output();

  cout << "Adding matrices n into m \n";
  m = m + n;
```

vubookshoppk.com

```
  m.output();

  cout << "Calling m + m + 4 \n";
  m = m + m + 4;
  m.output();

  cout << "Calling m += n \n";
  m += n;
  m.output();

  cout << "Calling m = m - n \n";
  m = m - n;
  m.output();

  cout << "Calling m = m - 4 \n";
  m = m - 4;
  m.output();

  cout << "Calling m -= n \n";
  m -= n;
  m.output();

  m.transpose();

  Matrix c;

  cout << "Calling c = m * n \n";
  c = m * n;
  c.output();

  cout << "Calling c = c * 4.0 \n";
  c = c * 4.0;
  c.output();

  cout << "Calling c = 4.0 * c \n";
  c = 4.0 * c ;
  c.output();

  cout << "Testing stream extraction \n";
//  cin >> c;

  cout << "Testing stream insertion \n";
 // cout << c;

  cout << "Writing into the file d:\\junk.txt \n" ;
  ofstream fo("D:/junk.txt");
  fo << c;
  fo.close();

  cout << "Reading from the file d:\\junk.txt \n";
```

vubookshoppk.com

```
    ifstream fi("D:/junk.txt");
    fi >> c;
    fi.close();

    cout << c;

    system("PAUSE");
    return 0;
}
```

The output of the program is:

```
Taking the input for m(4,5) and n(5,4)
Input Matrix size: 4 rows by 5 columns
Please enter 5 values separated by spaces for row no.1:   1.0  2.0  3.0  4.0  5.0
Please enter 5 values separated by spaces for row no.2:   7.0  5.5  2.3  2.0  1.0
Please enter 5 values separated by spaces for row no.3:   3.3  2.2  1.1  4.4  5.5
Please enter 5 values separated by spaces for row no.4:   9.9  5.7  4.3  2.3  1.5
Input Matrix size: 5 rows by 4 columns
Please enter 4 values separated by spaces for row no.1:  11.25  12.25  13.25  14.25
Please enter 4 values separated by spaces for row no.2:  25.25  50.50  75.75  25.50
Please enter 4 values separated by spaces for row no.3:  15.15   5.75   9.99  19.90
Please enter 4 values separated by spaces for row no.4:  25.50  75.75  10.25  23.40
Please enter 4 values separated by spaces for row no.5:  50.50  75.50  25.25  15.33
Displaying the matrix m(4,5) and n(5,4)

[   1.00   2.00   3.00   4.00   5.00
    7.00   5.50   2.30   2.00   1.00
    3.30   2.20   1.10   4.40   5.50
    9.90   5.70   4.30   2.30   1.50 ]

[  11.25   12.25   13.25   14.25
   25.25   50.50   75.75   25.50
   15.15    5.75    9.99   19.90
   25.50   75.75   10.25   23.40
   50.50   75.50   25.25   15.33 ]

Taking the transpose of matrix m(4,5)
Displaying the matrix m(5,4) and n(5,4)

[   1.00   7.00   3.30   9.90
    2.00   5.50   2.20   5.70
    3.00   2.30   1.10   4.30
    4.00   2.00   4.40   2.30
    5.00   1.00   5.50   1.50 ]

Adding matrices n into m

[
```

vubookshoppk.com

```
[ 12.25  19.25  16.55  24.15
  27.25  56.00  77.95  31.20
  18.15   8.05  11.09  24.20
  29.50  77.75  14.65  25.70
  55.50  76.50  30.75  16.83 ]
```

Calling m + m + 4

```
[ 28.50   42.50   37.10   52.30
  58.50  116.00  159.90   66.40
  40.30   20.10   26.18   52.40
  63.00  159.50   33.30   55.40
 115.00  157.00   65.50   37.66 ]
```

Calling m += n

```
[ 39.75   54.75   50.35   66.55
  83.75  166.50  235.65   91.90
  55.45   25.85   36.17   72.30
  88.50  235.25   43.55   78.80
 165.50  232.50   90.75   52.99 ]
```

Calling m = m - n

```
[ 28.50   42.50   37.10   52.30
  58.50  116.00  159.90   66.40
  40.30   20.10   26.18   52.40
  63.00  159.50   33.30   55.40
 115.00  157.00   65.50   37.66 ]
```

Calling m = m - 4

```
[ 24.50   38.50   33.10   48.30
  54.50  112.00  155.90   62.40
  36.30   16.10   22.18   48.40
  59.00  155.50   29.30   51.40
 111.00  153.00   61.50   33.66 ]
```

Calling m -= n

```
[ 13.25  26.25  19.85  34.05
  29.25  61.50  80.15  36.90
  21.15  10.35  12.19  28.50
  33.50  79.75  19.05  28.00
  60.50  77.50  36.25  18.33 ]
```

Calling c = m * n

```
[ 5117.55  8866.42  4473.54  3066.94
  7952.36 15379.14  7884.15  5202.50 ]
```

vubookshoppk.com

```
[  4748.18   8540.74   7566.73   3570.75
   3386.23   5949.35   4280.89   2929.51 ]
```

Calling c = c * 4.0

```
[  20470.19  35465.70  17894.15  12267.75
   31809.46  61516.55  31536.59  20810.01
   18992.71  34162.97  30266.91  14283.00
   13544.91  23797.41  17123.54  11718.05 ]
```

Calling c = 4.0 * c

```
[  81880.76 141862.80  71576.62  49071.00
  127237.84 246066.20 126146.34  83240.04
   75970.86 136651.88 121067.65  57132.02
   54179.64  95189.64  68494.16  46872.18 ]
```

Testing stream extraction
Testing stream insertion
Writing into the file d:\junk.txt
Reading from the file d:\junk.txt

```
[  81880.76     0.81     0.62     0.00
  127237.84     0.20     0.35     0.04
   75970.86     0.88     0.66     0.02
   54179.65     0.65     0.16     0.18 ]
```

Press any key to continue . . .

vubookshoppk.com

Lecture No. 45

Reading Material

Lecture 43 - Lecture 44

Summary

- Example (continued)
  - Assignment Operator Function
  - Addition Operator Function
  - Plus-equal Operator Function
  - Overloaded Plus Operator Function
  - Minus Operator Function
  - Multiplication Operator Function
  - Insertion and Extraction Operator Function
  - exercise
- Rules for Programming
- Variables and Pointers
- Arrays
- Loops and decisions
- Classes and Object
- Garbage Collection
- Truth Table
- Structured Query Language

vubookshoppk.com

Example (continued)

This is a sequel of the discussion on 'matrix class' made in the previous lectures.

Assignment Operator Function

Before going into minute details, we will talk about the assignment operator of this class. This operator occupies very important place in the field of programming. When we want to write code like *a = b;* where *a* and *b* both are matrices, the assignment operator attains a critical role as our class does dynamic memory allocation. Here we don't know whether the size of *a* and *b* will be the same or not. If these are of different size, then it will be better to reallocate the memory. So it warrants the existence of some checks and balances. At first, we look at the declaration line of the assignment operator, it is written below.

```
const Matrix & operator = (const Matrix &m);
```

The declaration line states that it must return a reference to a matrix. Why do we want to return a matrix? The return of the matrix is necessary to enable us write the statement i.e. *a = b = c;* We know, in C and C++ languages, every expression has a value of its own. Now if we write *a = b ;* it will mean that the whole action will have a value. This value itself will be a reference to a matrix. On the other hand, we have made it *const*. In other words, the reference that is being returned is a constant.
Whenever we return a reference of a thing that thing can become on left-hand side of an assignment statement, which can lead to some very funny behavior. If we write *(a = b) = c ;* It will result in the execution of the statement, *a = b.* It will return a value that is a reference to a matrix. This reference will be assigned the value of *c*. this means that some minor things can take place. We will like to have parentheses only on right hand side. It is advisable not to do an assignment to the reference that is being returned. To avoid this, we write *const* with it. Thus, we get the efficiency as reference is being returned. We also enjoy safety due to the fact that no value can assign to this reference in the same statement. Due to the reference returned, we can write the statement like *a = b = c ;*

Let's have a look on the next implication i.e. the size of the matrix. Whenever there is dynamic memory allocation in a class, we have to check against self-assignment. Self-assignment means statements like *a = a ;*. If we take ordinary variables, say integer, writing *i = i ;* is quiet right. But if we write something like *a = a ;* it will be possible to free the memory of object on the left hand side as it is doing some dynamic memory allocation. Now we can assign new memory and copy the right hand side in it. If we write *a = a;* it will lead to a very tricky situation. This means that we, at first, delete it (the left-hand side), as right hand side is the same object with same memory, so it is also deleted. Then, we try to copy the right hand side that has been deleted. So we must check against self-assignment. While dealing with the code of the assignment operator, we will first check whether it is for the self-assignment or not. To ascertain it, we will write the following line.

```
if( &m != this)
```

This way, the self-assignment check has been carried out. In case of not finding self-assignment, the programmer will have to do further process..
After checking the self-assignment, the program checks the size of both the matrices and sees whether these are the same or different. If their size is same, then it will copy the matrix of right hand side element-by-element in the matrix on left-hand side. But if their size is different in terms of number of rows or columns, then we have to create a new matrix for the left-hand side. This code is similar to the one that we wrote in the constructor. We free the memory and reallocate the memory of the correct size. This size is equal to the size of the matrix on right hand side. After re-allocating the memory, we readjust the number of rows and columns of the lef- hand side object. So we write it as

```
numerous = m.numRows ;
numCols = m.numCols ;
```

While defining rows and columns, we execute a *for* loop and copy the elements of right hand side at corresponding positions on left hand side. This way, we define the assignment operator, which can be used in different functions. Here, we can write

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

statement like *a = b which will* work properly. The same thing is applied to our main code when we come to a client function of the class. The code of the function of assignment operator is written as below.

```
const Matrix & Matrix :: operator = ( const Matrix & m )
{
    if( &m != this)
    {
        if (numRows != m.numRows || numCols != m.numCols)
        {
            delete [] elements;
            elements = new (double *) [m.numRows];
            for (int  i = 0; i < m.numRows; i++)
            elements[i]=new double[m.numCols ];
        }
        numRows = m.numRows;
        numCols = m.numCols;
        for ( int i=0; i<numRows; i++)
    {
            for(int j=0; j<numCols; j++)
            {
            elements[i][j] = m.elements[i][j];
            }
        }
    }
    return *this;
}
```

vubookshoppk.com

Addition Operator Function

Now we will discuss the addition operator. We have discussed a variety of addition operators. We come across one of these while writing *a + b* where *a* and *b* are matrices. While adding two matrices, it is ensured that these are compatible. It means that the matrices are conformable for addition. Their number of rows and columns should be equal. The code, we have just, written is very simple. It first checks whether the matrices are compatible. If so, it does the element-to-element addition. If these are not compatible, it returns the old matrix. Here the thing to remember is, what is being returned? The addition operator returns a new matrix after adding two matrices. The matrices, which were added, remain the same. So, if we add two matrices *a* and *b*, these will remain as it is and addition operator will return a resultant matrix by adding them. Therefore, if we find the matrices are compatible, a new matrix is defined. Having defined the new matrix, we can apply some tricks to it. We can define the new matrix by using copy constructor. So a complete matrix will be copied to it. This is reflective from the following statement.

```
Matrix temp (m) ;
```

When we a copy of one matrix, i.e. *temp*, the elements of the other matrix are added to it. We do this with a loop. After this, temp is returned. This temporary matrix (temp) which was created in the addition operator, is returned. However, its reference can not be returned. Here we have to return a matrix, ignoring the problem of

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

efficiency. So this whole matrix will be copied on the stack and assigned wherever it is needed. If we have written *c = a + b* ; it will be assigned to *c*. The things where the reference or matrix is being returned, should be carried out carefully. It is important for us to know what thing should be returned to where, and what is its usage and behavior? The code of the addition operator function is written in the program as below.

```
Matrix Matrix::operator + (  Matrix &m ) const
{
        // Check for conformability
        if(numRows == m.numRows && numCols == m.numCols)
        {
                Matrix temp(*this);
                for (int  i = 0; i < numRows; i++)
                {
                        for (int j = 0; j < numCols; j++)
                {
                temp.elements[i][j] +=  m.elements[i][j];
                        }
                }
                return temp ;
        }
        Matrix temp(*this);
        return temp;
}
```

Plus-equal (+=) Operator Function

Now we will discuss the += operator. Whenever a programmer writes *a += b*, he will come across a different scenario as compared to the one witnessed in the case of the addition operator. In a way, now '*a*' itself is being changed. So if *a* is going to change, we can return a reference to *a*. The conformability check remains the same as in ordinary addition. That means both matrices must have the same number of rows and columns. There is no need of creating a temporary matrix. Here we can return reference to left-hand side matrix. Here one finds that there is reuse and efficiency in the code. The += operator is defined as under.

```
const Matrix & Matrix::operator += (Matrix &m)
{
        *this = *this + m;
        return *this;
}
```

Overloaded plus Operator Function

Next concept to be discussed the overloading of the overloaded plus (+) operator. It is very simple. If we want to add a *double* variable in a matrix, there will be no problem of conformability. All we need to do is add a value to every element. Here we are doing *a + d* (*a* is a matrix while *d* is a *double*). Here, *a* will not change. A new matrix will be returned by adding the value of *d* to the elements of *a*. So it is not returning a reference, but a matrix. If it is returning a matrix, it must return a matrix that is created inside this function. Thus, we can use copy constructor and write

vubookshoppk.com

```
Matrix temp (* this) ;
```

In this matrix *temp,* we add the value of *d* by a nested loop and return it. This way, the addition of a matrix with a *double* variable is defined. The code of it is given below.

```
Matrix Matrix::operator + ( double d ) const
{
        Matrix temp(*this);
        for (int  i = 0; i < numRows; i++)
        {
                for (int j = 0; j < numCols; j++)
                {
                                temp.elements[i][j] +=  d;
                }
        }
        return temp ;
}
```

Next function is *d + a* (i.e. double variable + matrix). There is a *double* variable and not a matrix on left hand side, leaving no option for having it as a member function. It is a friend function and defined outside the class. We don't use the scope resolution operator (::) with it. It is defined as an ordinary stand-alone function. It still returns a matrix .So it is written as

```
Matrix operator + (double d, Matrix &m)
```

Two arguments are passed to it that are the variables on left and right side of the operator. The remaining code is almost the same as that of *a + d*, and is written as below.

```
Matrix operator + (double d, Matrix &m)
{
        Matrix temp(m);
     for(int i=0; i< temp.numRows; i++)
        {
        for(int j=0; j<temp.numCols; j++)
                {
                temp.elements[i][j] += d;
        }
         }
        return temp;
}
```

Minus Operator (-) Function

The same discussion of plus (+) operator applies to the minus (-) operator as both are identical. We see the difference between these operators when we do *a + d*. In case of addition, it will be the same as that of d + a. However, while dealing with minus case, the result of *a - d* will be obviously different from the result of *d - a*.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

Multiplication Operator (*) Function

The most complicated operator out of all these arithmetic manipulators for matrices is the multiplication (*) operator. How do we multiply two matrices together? We have already discussed it while defining matrices. We have discussed that the size of the resultant matrix will be the number of rows of the first matrix multiplied by the number of columns of second matrix. We also have discussed the method to calculate the element of the resultant matrix. Obviously, before doing this, we check the conformability of the matrices for multiplication. The code of the function for * operator is defined as below.

```
Matrix Matrix::operator* ( const Matrix& m)
{
        Matrix temp(numRows,m.numCols);
        if(numCols == m.numRows)
        {
                        for ( int i = 0; i < numRows; i++)
                        {
                                        for ( int j = 0; j < m.numCols; j++)
                                {
                                temp.elements[i][j] = 0.0;
                                for( int k = 0; k < numCols; k++)
                                {
                                        temp.elements[i][j]     +=      elements[i][k]     *
m.elements[k][j];
                                }
                                        }
                        }
         }
        return temp;
}
```

The multiplication of a matrix with a *double* is nothing more complicated as that of doing addition of a matrix with a *double*. So code used in both cases is similar. In the case of division of a matrix by a double, the only thing that differs is that while dividing a matrix with a *double*, we have to check the division by zero. After having that little check, we divide the matrix. Without going for some complex method, we simply return the original matrix if it encounters a division by zero. Mathematically speaking, it is not correct. We should actually throw an exception so that program should stop in such a case. But we do not throw an exception and return the original matrix without trying to divide it by zero. So our program does not generate a run time error. There may be logical errors. So we have to be careful in such a case.

Insertion (<<) and Extraction (>>) Operator Function

The last set of functions that we have defined is the stream insertion (<<) and extraction (>>) operators. These operators may be taken as the example of code reuse. We have already defined input and output functions for this class. These input and output can handle matrices with files or on screen. Now we want an operator to write *cin >> m* ; where *m* is an object of type Matrix. It means that we have to read from the keyboard and store these values in the matrix *m*. Inside the code, we can reuse the input function and write *m.input* in the function body. This is the overloaded stream

vubookshoppk.com

extraction operator. The only difference is that we have to return a reference to the stream. Thus it is a two- line function and is a good example of code reuse. We have written the input function, which can be used here. Same thing applies if we have to take input from the file and put it into matrix *m*. We have declared these functions as friend functions and in the following their code is written

```
istream & operator   >> ( istream & is, Matrix & m)
{
        m.input(is);
        return is;
}
```

We will now use the input function, written to take input from file.

```
ifstream & operator   >> ( ifstream & is, Matrix & m)
{
        m.input(is);
        return is;
}
```

Similarly, the pair of output functions can be reused to overload the stream insertion operator.

```
ostream & operator   << ( ostream & os, Matrix & m)
{
        m.output();
        return os;
}
```

And for the file output the code is as follows.

```
ofstream & operator   << ( ofstream & os, Matrix & m)
{
        m.output(os);
        return os;
}
```

Exercise

Now you should study and understand this whole code of the class and use it. Its use is very simple. You can write the main function and in it write

```
Matrix m (3,3) ;
```

When you will execute it, a matrix of three rows and three columns will be created and values of it will be zero. To display this matrix on the screen, you can write

```
m.output ;
```

It will display a properly formatted matrix on the screen. Similarly you can define other matrices and can get their values from the keyboard. You can multiply them and

vubookshoppk.com

see that multiplication is done only if the matrices are conformable for multiplication. Similarly, addition will work only if the matrices are conformable for addition. You can write a little test program. You should also try to extend the class by adding new functions and features into it. In the code, there are not proper error messages. You can write code to do more error checking and to display proper error messages wherever an error encounters.

It is very simple to change the whole class from *double* to *int*. More complicated one would may be used to write a template for this class. In the class, wherever there is *double*, you will write <T> there and on the top, there will be template <class T>. The remaining things will almost look identical. You will have to take care in friend functions. So there is a lot of stuff you can do with it.

Review

Now we will review the different topics of the course briefly and some discussion in respect of languages and programming. In the beginning of the course, we came across a few programming guidelines. We have read about design recipe. Then we went on and developed the way of thinking.
To begin with the review of the previously discussed subjects, we will now talk about the rules of programming.

Rules for Programming

We need simply three constructs to solve any problem.

1) Things should be executed sequentially. That means the statements are executed in a sequence i.e. the second statement follows the first and so on.

2) We need to have a decision. The decision means if something is true, the program executes it. Otherwise, it tries doing something else. So there is simple decision or if-else decision.

3) The third construct is loop, which is a repetition structure that performs the same task repeatedly with different values.

So the availability of sequences, decisions and loops can help us write any program. The code, we write, should be short and concise. It need to be self-contained and understandable. Comments should be placed liberally. The comments should explain the logic, not the mechanics. Try to avoid fancy programming. The indentation of the code has no means programmatically as it does not mean any thing at all. What actually matters is how you structure the code using braces and semicolons i.e. the structure of the language. Otherwise, C and C++ are free-format languages. We can write the whole code in a single line. But it will be very difficult to read. We format our code so that it could be read easily. So indentation is for us, not for the compiler. Similarly, any language does not dictate it. On the other hand, if we put our code inside braces and blocks, it will ensure a logical syntax.

Variables and Pointers

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

After constructs, the concept of variables and pointers holds very important position in programming. The variable is a name for a value. It is like a label on a box in the memory, which contains a value. We can use this label to manipulate the value, instead of using the address of the memory that contains the value. There are different types of variables.
We discussed earlier, the pointers are much more specific to C and C++. A pointer is an address of a location in the memory. We also have talked about their manipulations.

Arrays

An array is a type of data structure. We use an array to store multiple values of the same data type. In C, C++ and FORTRAN languages, the arrays are of the same data type i.e. every element of the array is of the same data type. There can be an array of integers, an array of characters and so on. We cannot have elements of different types in an array. There are the languages in which we can have arrays of mixed-type. FoxPro is the quotable example in this regard. We can store in a variable whatever we want. For example if we write *a = 3* then *a* is a numerical value. On other hand suppose, if we write *a = "This is a string"*. Here '*a*' becomes a character string. Similarly in Visual Basic, there is a data type, called *variant* that can store data of all kinds. So remember that whenever we talk of arrays and variables, different languages behave differently. There are no hard and fast rules.

Loops and Decisions

The loops and decisions are 'bread and butter' for a programmer.
While talking about decisions, we read that in C and C++ languages, there is a simple *if* statement. There is also '*if-else* statement'. We can write a big structure by using the nested *if-else* statements. There is also *switch* statement. These statements (if, if-else and switch) are language specific. Almost all the modern programming languages provide a decision structure. The exact syntax of which can be got from language reference.

While talking about repetition structure, we come across the concept of loops. The loops are of three different types in C ++. These include *while*, *do-while* and *for* loop. There is a subtle difference between them. While using the *while* loop, if its condition is false at the start, its body will not execute even once. In other words a *while* loop executes zero or more times. On the other hand, if we write a *do-while* loop, the block of code written after the *do* will execute at least once. So a *do-while* loop executes one or more times. The *for* loop is more like the *while* loop. It will execute zero or more times. Every loop has a basic structure that is independent of the language. There is some initialization, a condition that is tested for the execution of the loop. Then there is the body of the loop in which it performs its task. These are almost same in the languages. But the syntax is particular to the language.

Classes and Objects

In this course, we discussed, only the concept of, classes and objects. The study of rudiments of classes and objects can help us understand the difference between implementation and interface besides comprehend the concept of the encapsulation. We combine the data and code to form an object. It is a new type of variable, a user-defined data type. It not only has its data structure but also the code that manipulates

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

the data. The major advantage of data hiding and encapsulation is that it makes every thing tested, debugged and ready to use. When we come in the main program, which uses these classes or objects, our code becomes very simple. We can reuse this code repeatedly. When we put all of this together, the concept of doing object-oriented programming becomes clear. How can a class be made from another class? We will talk about polymorphism in the course of object oriented programming. It can determine what function is to call at the execution time of the program not at the compile time. These are very important and powerful methods. There will be whole idea of thinking objects. Here we only covered mechanics. When we were talking about mechanics, we have to understand how can we implement a member function and a member operator. We use the sequences, decisions and repetition structures while writing the member or friend functions. So we build on our previous knowledge and introduce the concepts of classes and objects.

Garbage Collection

The whole concept of using objects and their notation which is *object.member*, where member could be a data type or a function, that is what we have been exercising. We also mentioned that we could have pointers to objects. During the manipulation of the data variable or data member with pointers, we use the arrow (->) notation rather than the dot (.) notation. The concept of pointers is very important but quite limited to C and C++. The modern languages, for example JAVA, describe pointers as dangerous. We can go anywhere in the memory and can change a value. There is another problem with pointers, which is that these could be pointing to nowhere. For example, we allocate memory and de-allocate it there or in some other function, without reassigning a new memory to the pointer that was pointing to that memory. Thus, a dangling pointer is there that points to nothing. There is also reverse case of it that we assign a memory through a pointer where the pointer is destroyed, the memory remains allocated and is wasted. To address these things, there are only references in JAVA instead of pointers. JAVA gives the concept of garbage collection with the use of references. Due to this garbage collection, we are free from the headache of de-allocating the memory. We allocate and use the memory. When it is no longer in use, JAVA automatically deletes (frees) it through garbage collection. But in C and C++ languages, we have to take care of de-allocating the memory. In classes where we use dynamic memory, we have to provide destructors to free this memory. The languages keep evolving, new constructs will keep evolving in existing or new languages. So the foundations of our knowledge must be strong. We have to know what is programming. We have to know how can we take the essence of a problem by analyzing it. We should repeat the design recipe as many times as needed.

Truth Table

There are some areas where the decision structures become very complicated. Sometimes, we find it difficult to evaluate a complicated logical expression. Sometimes the logic becomes extremely complicated so that even writing it as a simple syntax statement in any language. It becomes complicated to determine what will be evaluated in what way. We know the concept of truth table. The truth tables are very important. These are still a tool available for analyzing logical expressions. We will read logic design in future, which is actually to do with chips and gates. How we put these things together. In logic design, there are certain techniques that are

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533

known as minimization techniques. These are used to make a big circuit with the use of minimum chips. These minimization techniques deal with Boolean algebra i.e. logic. These techniques are also used in programming. So we should keep breadth in our vision while maintaining a horizontal integration. We should always think outside the box. There is a way of thinking for us as programmers. We always look at problems, slice and dice them and come up with solutions. Programming as a skill is infact important. It helps us think, from a logical perspective. How can we do it is something else. We can get it from the reference books of the language or from online help in the compiler. This part that how can we do is always changing. New languages will be evolved for our help. On the other hand, what is to be done depends on our logical skills and fundamental knowledge. We have to develop this thing.

Structured Query Language

In the business world, most of the programming is database-oriented. In today's databases, like Oracle and SQL Server, a different kind of language is used. These are the languages that are called as structured query languages i.e. SQL. SQL, is so important that a standard has been developed for it. So there is an ANSI standard for this language. There is a major difference between SQL and the conventional language. The SQL by law says 'tell me what do you want and I will determine how to do it'. Whereas in our conventional languages like C or C++, we have to tell the languages what we want to do and how to do it. These are differentiated in the terminology like third generation languages and fourth generation languages.

There are optimizers built in those languages. Optimizers mean how a query or question can be executed more efficiently. In the same way, there are optimizers in our compilers. When we write the code, the compiler looks into it to determine how this code can be executed more efficiently. The modern compilers do a large optimization. Different software companies or computer manufacturers write the compilers. The standard is the same for writing compilers. The difference is that how much fast the executable version of a program executes and how much memory it uses, when compiled by different compilers. The speed and memory usage is the two yard sticks of output code.

The fundamentals are important. Keeping of a breadth of vision is also critically important. We have to constantly keep up with literature, keep up with new development, and experiment with more and more new tools.

Talking about languages is not that important. However, talking about programming is critically more important. If we have a sound fundamental knowledge, no new language can frighten us. We will never feel over powered by any new language. The fundamentals can become strong only by practicing more and experimenting to the maximum.

All VU books are available in hard copy and delivered to your doorstep. Order now!
Website: vubookshoppk.com | WhatsApp: 0326-0775533