

# **Introduction to Programming CS201**

|                      |     |
|----------------------|-----|
| 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 ..... | 269 |
| Lecture No. 24 ..... | 279 |
| Lecture No. 25 ..... | 292 |
| Lecture No. 26 ..... | 305 |
| Lecture No. 27 ..... | 319 |
| Lecture No. 28 ..... | 329 |
| Lecture No. 29 ..... | 345 |
| Lecture No. 30 ..... | 359 |
| Lecture No. 31 ..... | 369 |
| Lecture No. 32 ..... | 382 |
| Lecture No. 33 ..... | 395 |
| Lecture No. 34 ..... | 408 |
| Lecture No. 35 ..... | 422 |
| Lecture No. 36 ..... | 433 |
| Lecture No. 37 ..... | 444 |
| Lecture No. 38 ..... | 454 |
| Lecture No. 39 ..... | 469 |
| Lecture No. 40 ..... | 482 |
| Lecture No. 41 ..... | 497 |
| Lecture No. 42 ..... | 510 |
| Lecture No. 43 ..... | 520 |
| Lecture No. 44 ..... | 529 |
| Lecture No. 45 ..... | 553 |

## Lecture No. 1

### Summary

- o What is programming
- o Why programming is important
- o What skills are needed
- o Develop a basic recipe for writing programs
- o 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. “

## 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:

- o Critical reading
- o Analytical thinking
- o Creative synthesis

## 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

- o Paying attention to detail
- o Think about the reusability.
- o Think about user interface
- o Understand the fact the computers are stupid
- o 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 “

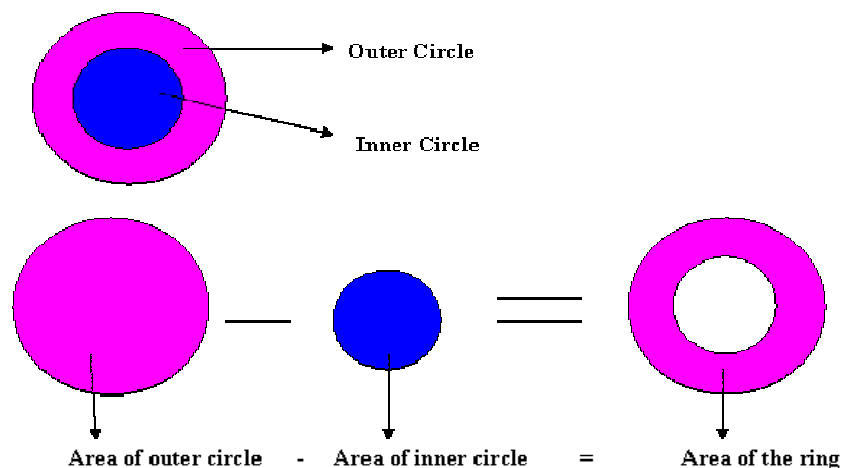
The grammar is correct but there is no meaning. Similarly, the sentence, "Mr. ABC sleeps thirty hours every day", is grammatically correct but it is illogical.

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.



### ***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.

### ***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.

### **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 elegantly 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 continues to say that:

"A programmer is also very much like an architect, a composer, 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. "

Hence to design a program properly, we must:

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

- o Don't assume on the part of the users

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

## 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

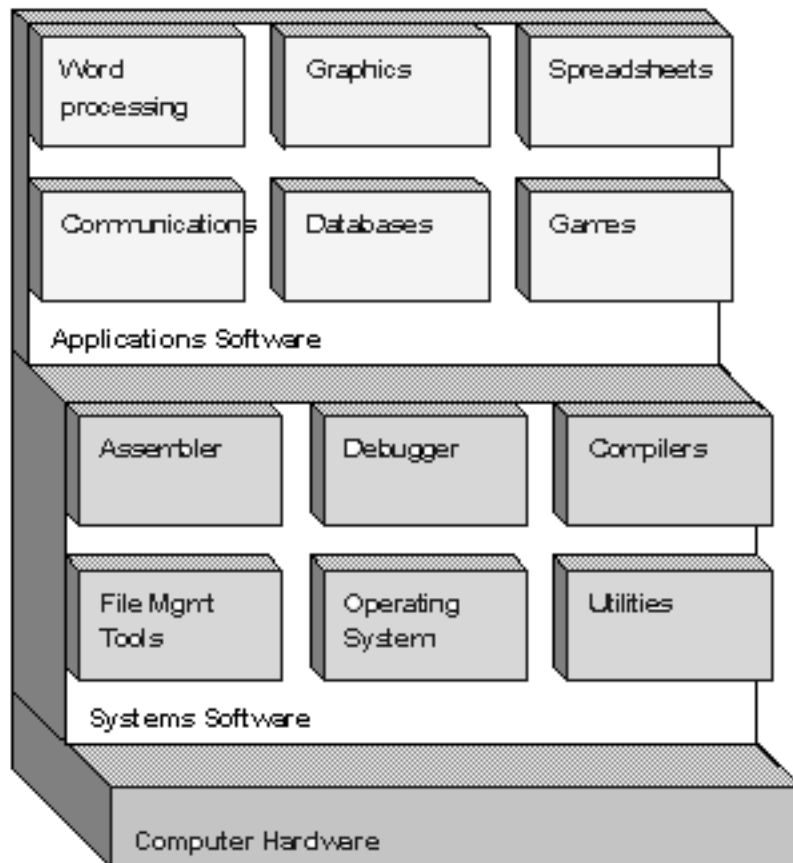
### ***Summary***

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

### ***Software Categories***

Software is categorized into two main categories

- o System Software
- o Application Software



## 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:

- o Operating system
- o Device drivers
- o 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)”

## 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

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

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

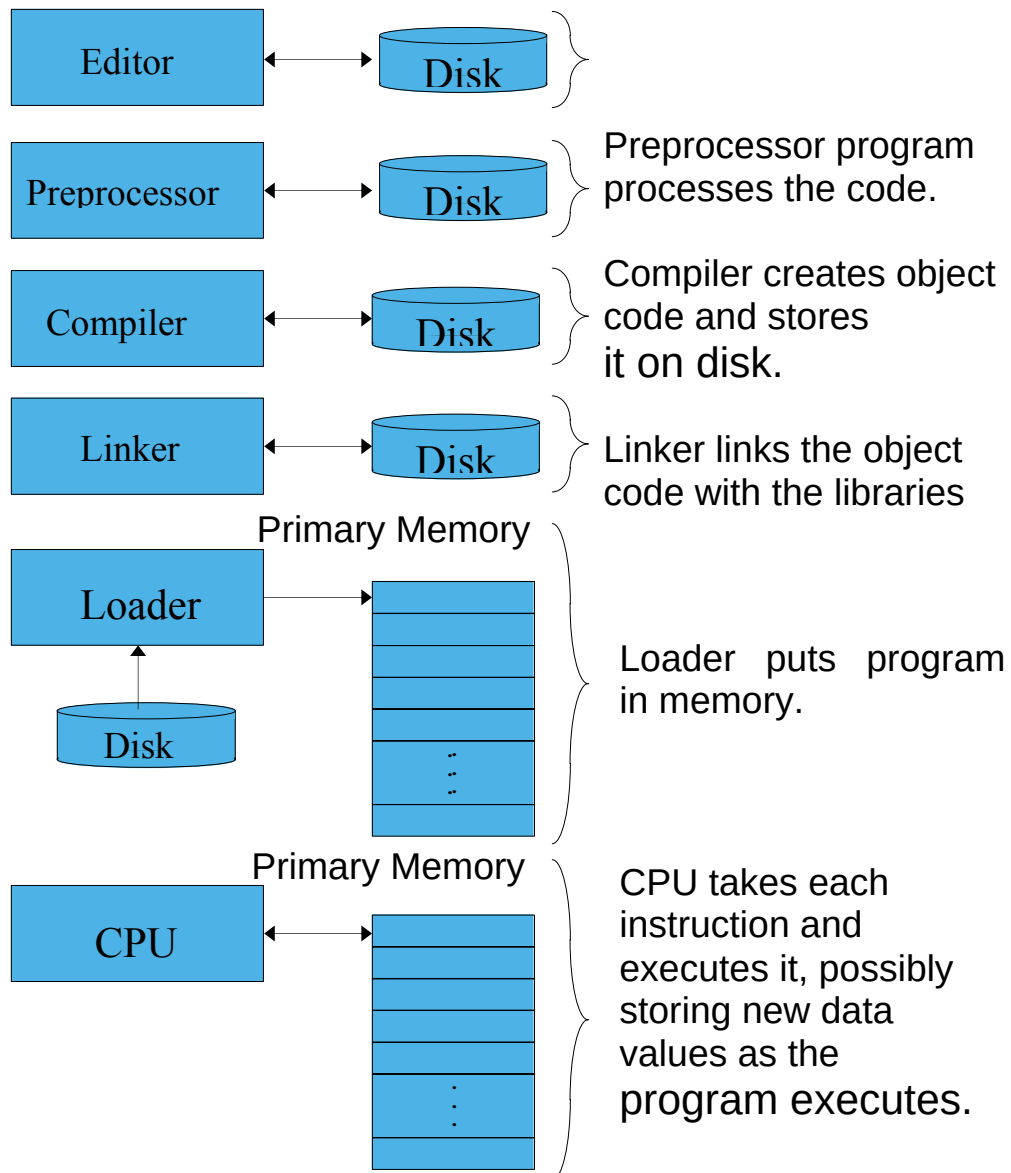
## Linker

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.



## 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

#### 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** will 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.

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**

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.

## 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 */
```

```
#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;
```

```

    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 | * | $x \times y$      | $x * y$  |
| Division       | / | $x \div y, x / y$ | $x / y$  |
| Modulus        | % | $x \bmod y$       | $x \% y$ |

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.

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.

#### TIP

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

## Lecture No. 4

### Reading Material

Deitel & Deitel – C++ How to Program

chapter 1  
1.22

### **Summary**

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

### **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

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:

```
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:

```
/* 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

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 2 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:

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 7,5,3,1 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.

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

**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
```

***Tips***

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

## Lecture No. 5

### Reading Material

Deitel & Deitel – C++ How to Program

chapter 2  
2.4, 2.5, 2.6, 2.19,  
2.20

### **Summary**

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

### **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

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.

|                          | 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.

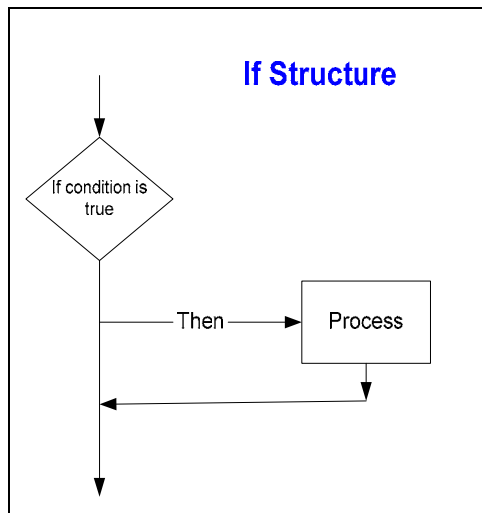
Don't confuse the assignment operator (=) with equal to operator (==). If we write single = in condition of *if* statement. For example, if we write *if ( x = 2 )*, the compiler will not give error. This means that it is not a syntax error. The conditional expression in *if* statement returns a value. 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.

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.

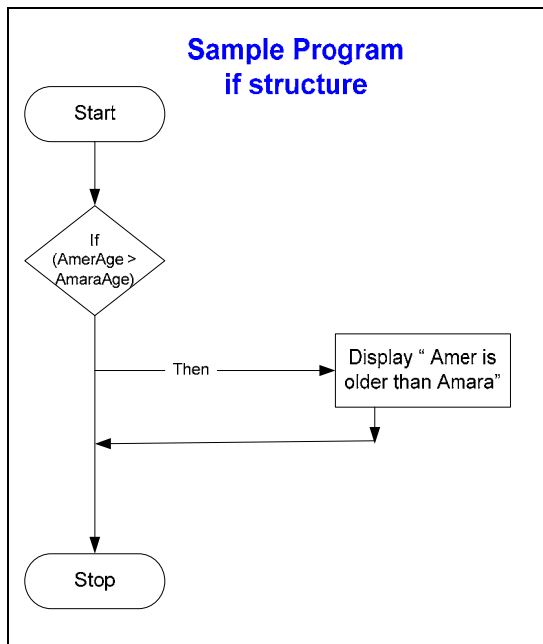
We can do this in one line while declaring a variable like `int x = 0;` This statement will declare a variable of name x with data type int and will assign a value 0 to this variable. Initializing a variable in this way is just a matter of style. You can initialize a variable on a separate line after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



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";
}

```

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)

```
{  
    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.

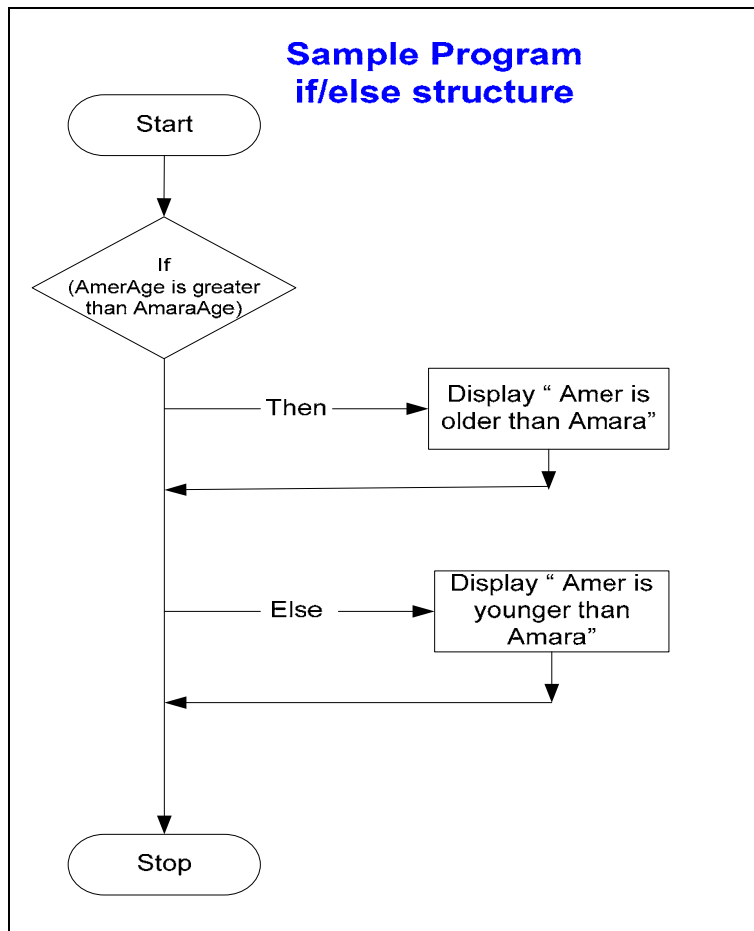
And in the comparison, we know that there are three conditions i.e. first value is greater than the second value, first value is less than the second value and first value is equal to the second value. Here in the above program construct the else part competes the greater than conditions and covers both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



## 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 &&<br>Expression 2 | Expression 1   <br>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         |

## Sample Program 2

### Problem statement

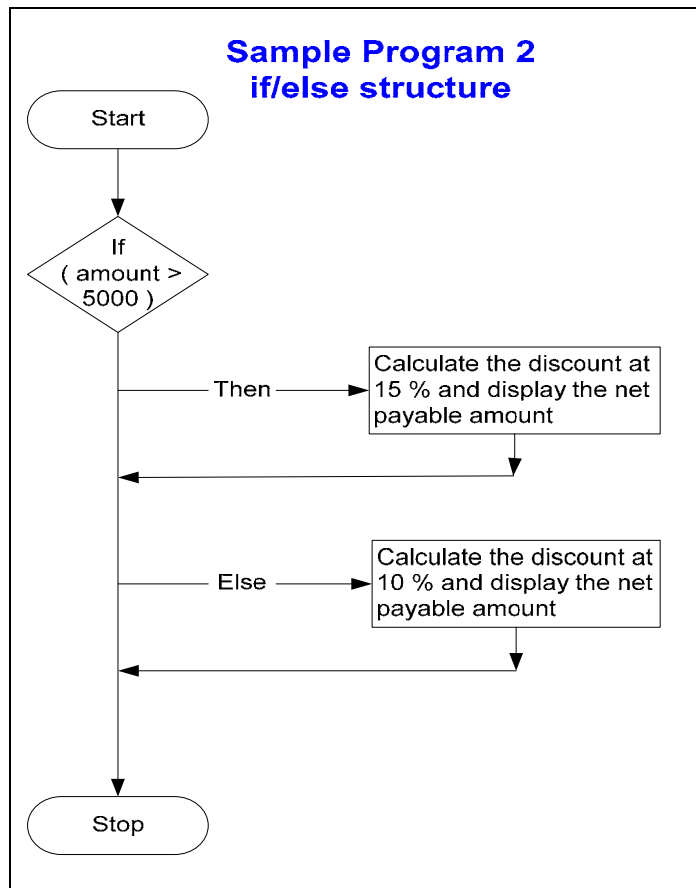
A shopkeeper announces a package for customers that he will give 10 % discount on all bills and if a bill amount is greater than 5000 then a discount of 15 %. Write a C program which takes amount of the bill from user and calculates the payable amount by applying the above discount criteria and display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

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.



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

|                                         |      |
|-----------------------------------------|------|
| 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

## 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

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

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

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.

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;
```

```

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  $(2 * (\text{number} / 2))$  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  $(\text{number} \leq \text{upperLimit})$  and in the *while* block the value of *number* is changing ( $\text{number} = \text{number} + 1$ ) 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  $\text{number} = \text{number} + 1$ ; 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:

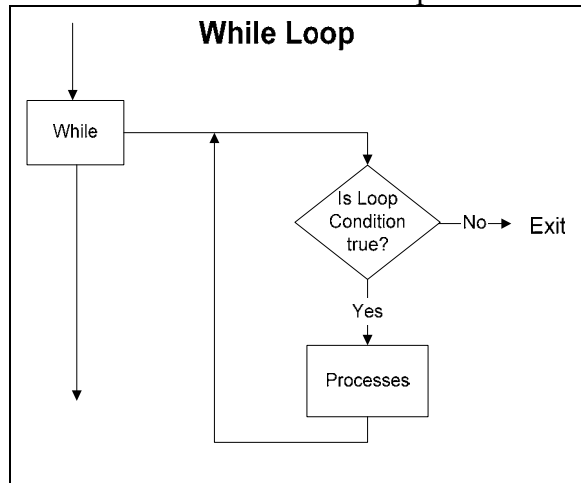
In the above example, if the user enters 0, as the value for upper limit. In the while condition we test  $(\text{number} \leq \text{upperLimit})$  i.e. *number* is less than or equal to *upperLimit* ( 0 ), this test return false. The control of the program will go to the next statement after the *while* 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

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

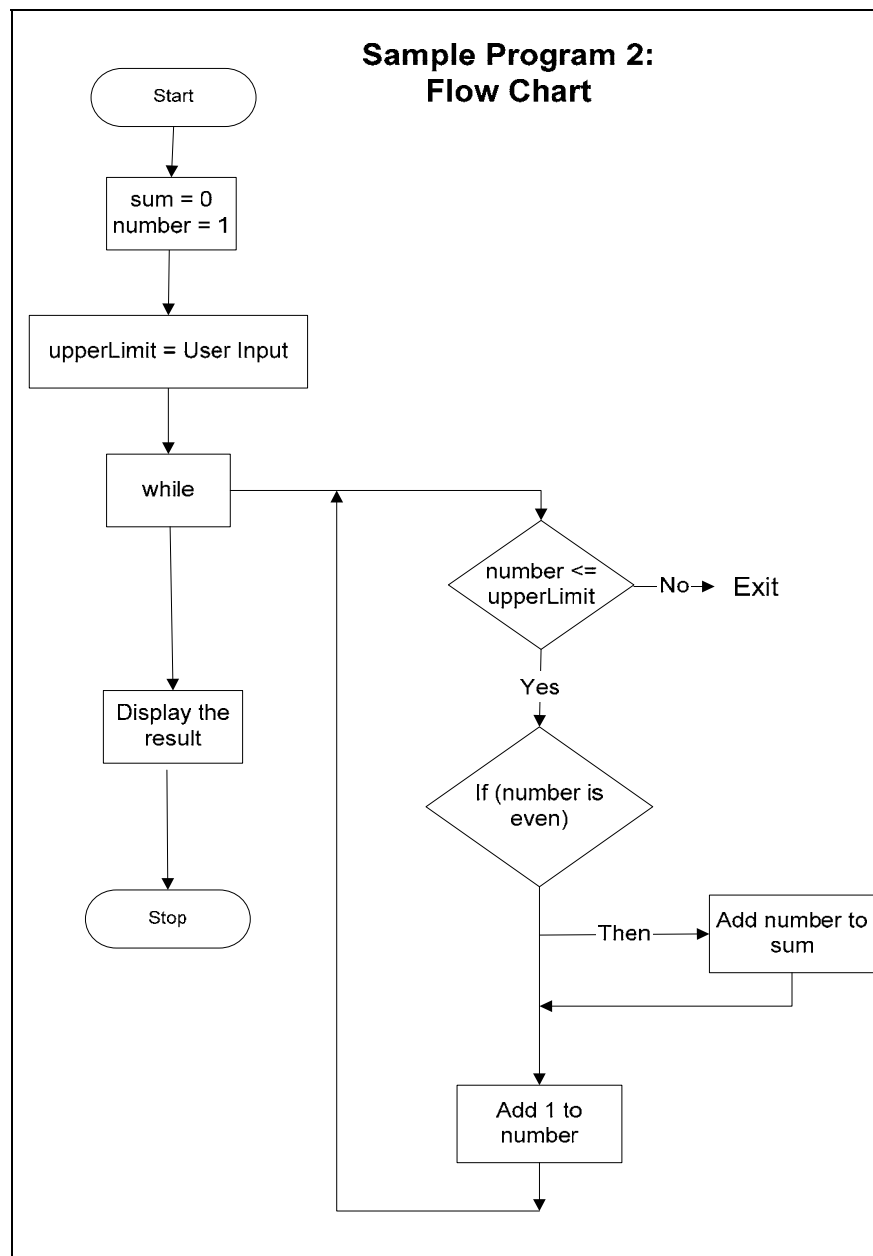
Flow Chart:

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:



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.

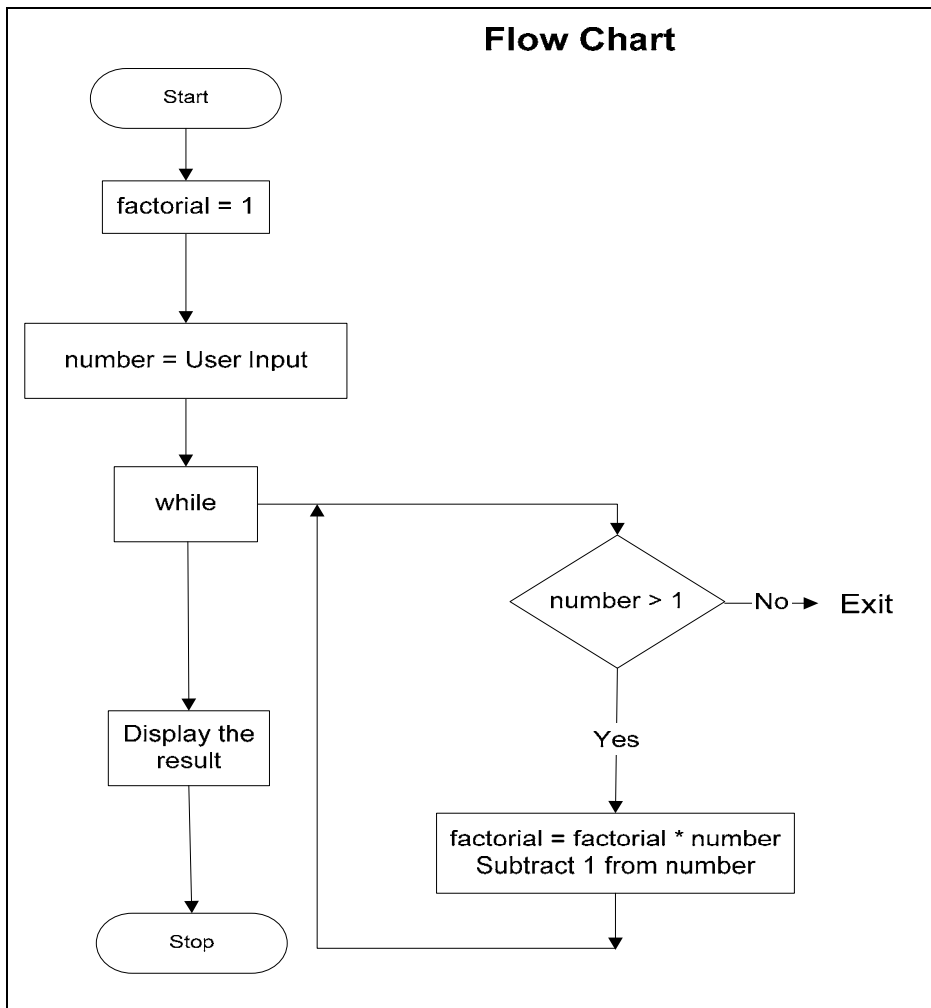
**Solution:**

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.



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;
}

```

}

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.

## Lecture No. 7

### Reading Material

Deitel &amp; 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

## 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 ) ;
```

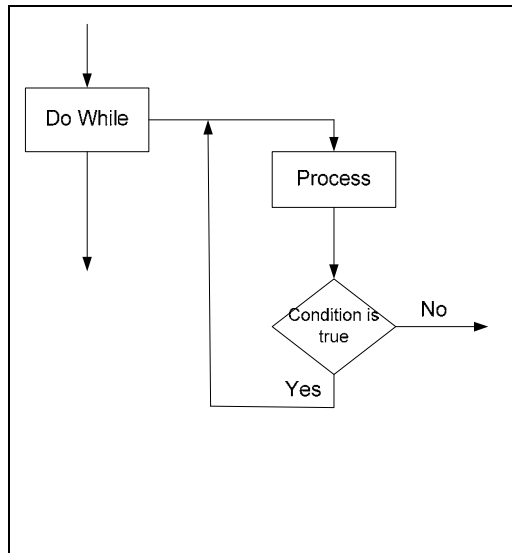
Here we see that the condition is tested after executing the statements of the loop body. Thus, the loop body is executed at least once and then the condition in *do while statement* is tested. If it is true, the execution of the loop body is repeated. In case, it proves otherwise (i.e. false), then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

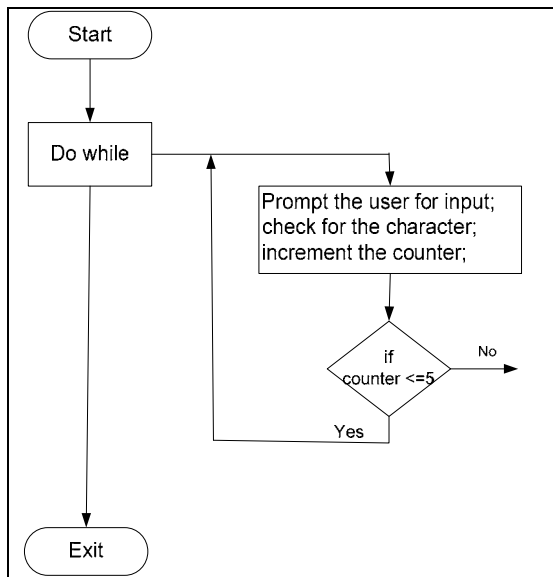
We declare the variable **c** of type *char*. The data type *char* is used to store a single character. We assign a character to a variable of *char* type by putting the character in single quotes. Thus the assignment statement to assign a value to a *char* variable will be as **c = 'a'**. Note that 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 ( $tryNum \leq 5$ ). If this condition is true, then the body of the *do* clause is repeated again. We do this only when the condition ( $tryNum \leq 5$ ) 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  $tryNum \leq 5$  becomes false, so we assign a value which is greater than 5 to **tryNum** after displaying the message. Now the

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);
  
```

```
}
```

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
```

## 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*.

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.

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
```

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
```

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

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 operand 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

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.  $\text{Sum} = 1^2 + 2^2 + 3^2 + \dots + n^2$ )

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;
```

```
    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

## Lecture No. 8

### Reading Material

Deitel &amp; Deitel – C++ How to Program

Chapter 2

2.16, 2.18

### *Summary*

- o Switch Statement
- o Break Statement
- o Continue Statement
- o Guide Lines
- o Rules for structured Programming/Flow Charting
- o Sample Program
- o 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" ;
```

```
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 : statementList1 ;
    case constant2 : statementList2 ;
        :
    case constantN : statementListN ;
    default : statementList ;
```

```
}
```

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.

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 downward. 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.

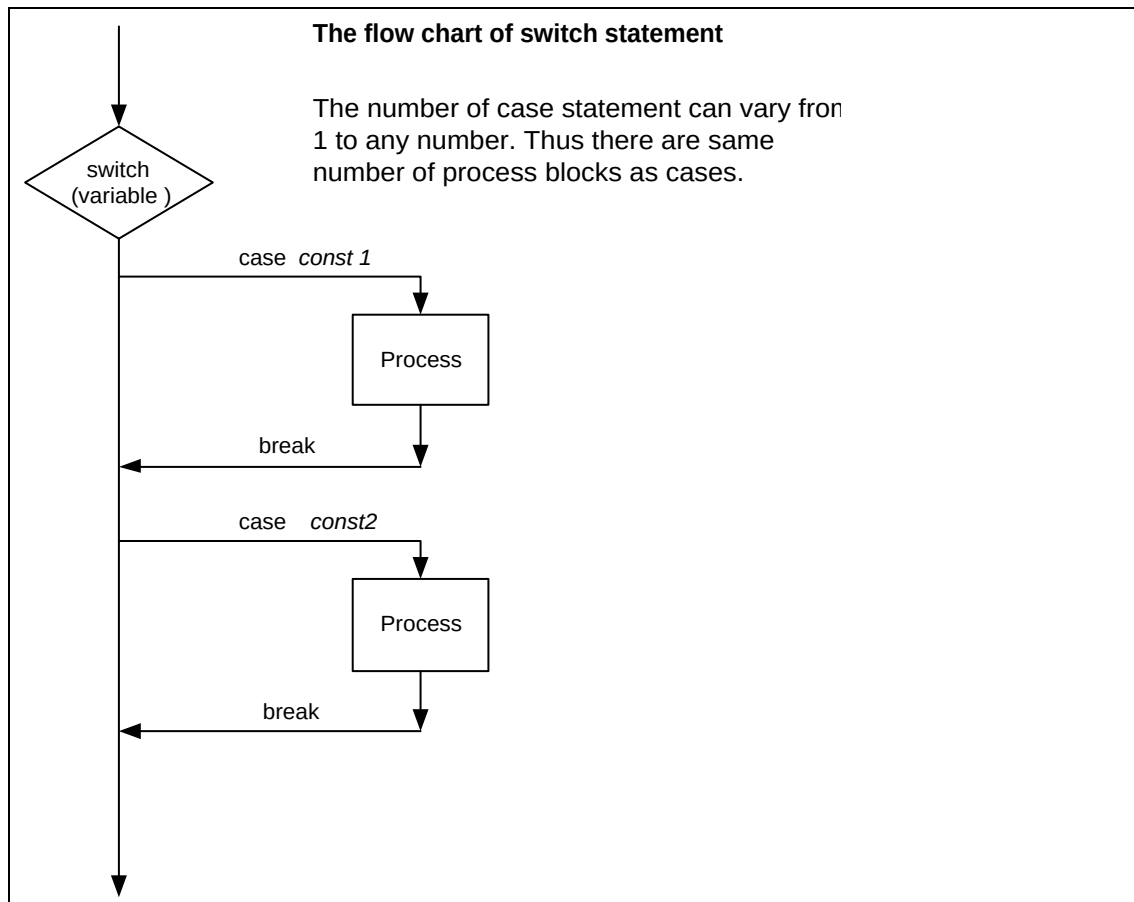
```
default : cout << "Please enter grade from A to D or F " ;
```

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;
}
```

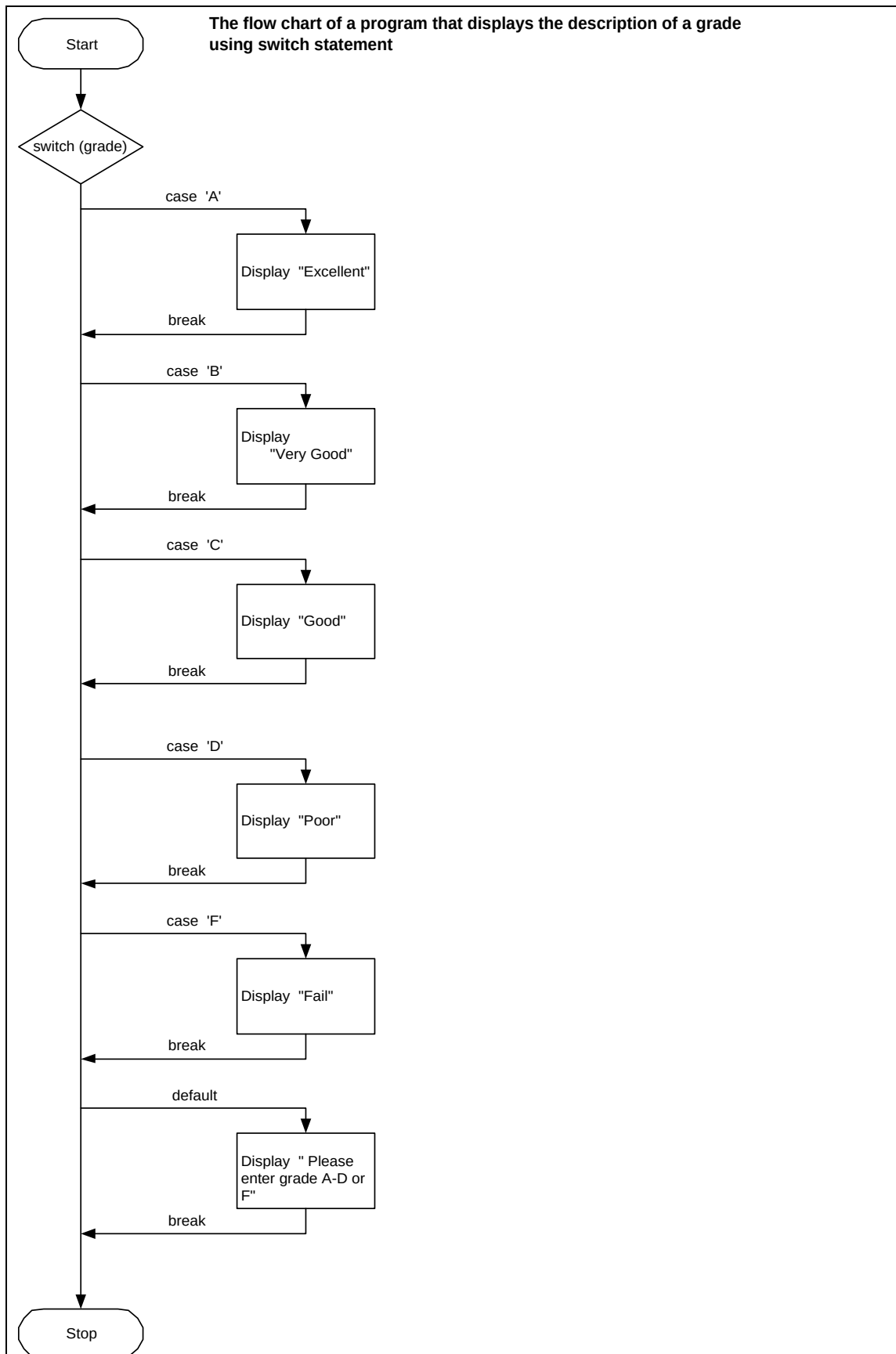
Thus, *break* statement can be used to jump out of a loop very quickly.

The flow chart of the *switch* statement is similar to *if* statement and is given below.



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.



The code of the program is given below.

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

```
# 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 ;
```

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 spaghetti 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

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

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

## 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***

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

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

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 then 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.
```

```
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;
}
```

### 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 ) ;
}
```

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.

## 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 data 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'.

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  $\text{Pi} * r^2$ . 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 ) ;
}
```

```

        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.

#### 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;
    }
}
```

## 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

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.

## 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

### 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).

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.

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.

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
```

```
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

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;
}
```

```
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:

```
#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);

void 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.

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:

$$n! = (n) * (n-1) * (n-2) * \dots * 2 * 1$$

Note that

$$n! = (n) * (n-1)!$$

$$\text{and } (n-1)! = (n-1) * (n-2)!$$

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

```
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.

‘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.

## 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

## 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



In the above figure, the memory chunk containing the array C is shown. On the first line, C[0] is written while on the 2<sup>nd</sup> 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 (C[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 5<sup>th</sup> 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

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 2<sup>nd</sup> 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*.

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 };
```

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.

## Sample Program 1

### Problem Statement:

**Write a program which reads positive 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 4<sup>th</sup> 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 ];
    int i, z;
```

```

do
{
    int z , i = 0 ;
    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.

```
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++ )  
    {  
        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

## 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 *i*th 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 " ;
}
```

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 100<sup>th</sup> 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 " ;
}
```

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 ++)
```

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

## 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

## 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

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 to 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

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
```

```

#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" ;
}

```

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 overwritten. 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.

## 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:

```
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 anything. 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

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 ;
}
```

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

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 ++ )
```

```
{
    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 |

## 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

### 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

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:

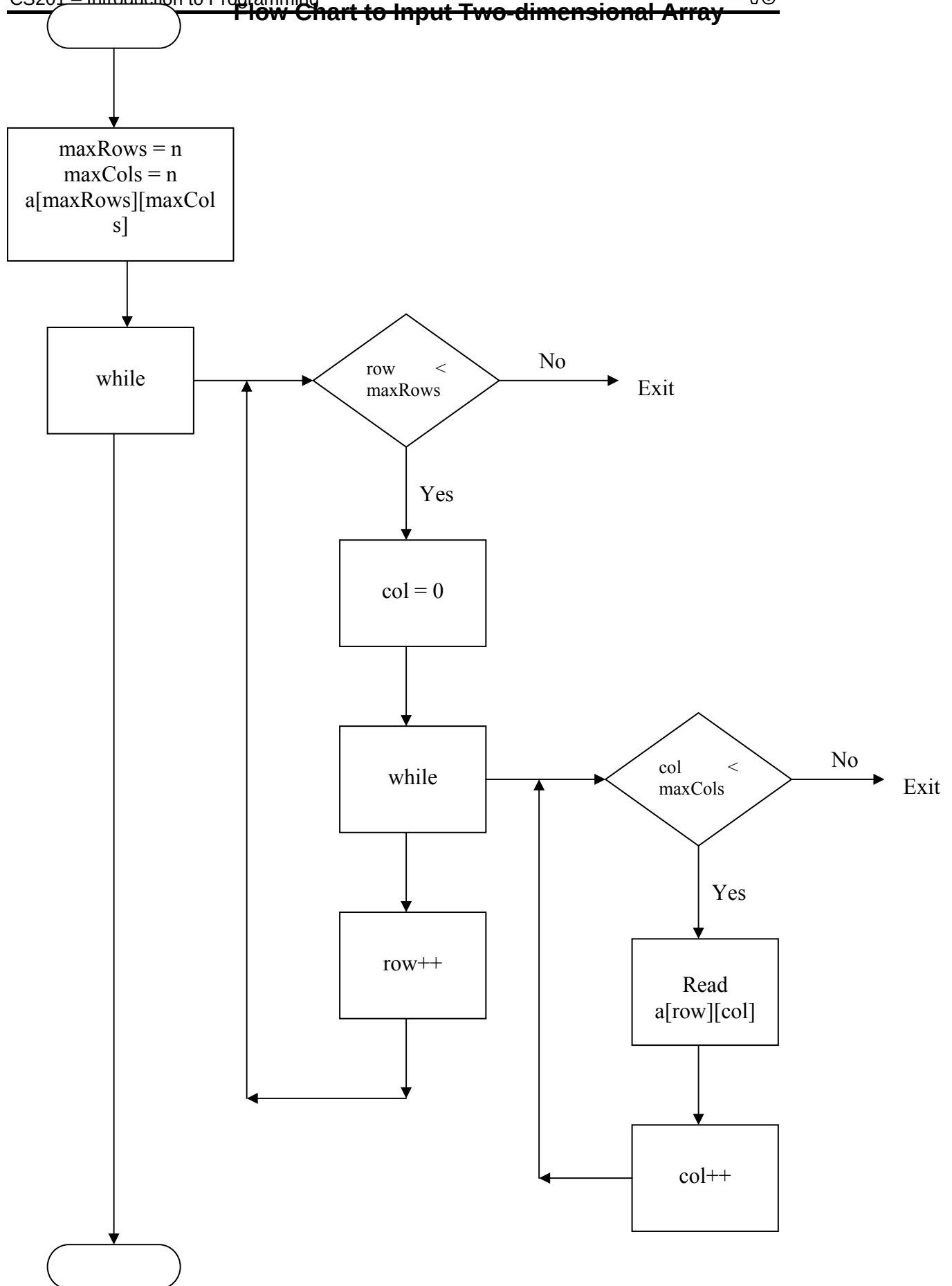
```
const int maxRows = 3;
const int maxCols = 3;
int row, col;

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**.

# Flow Chart to Input Two-dimensional Array



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';

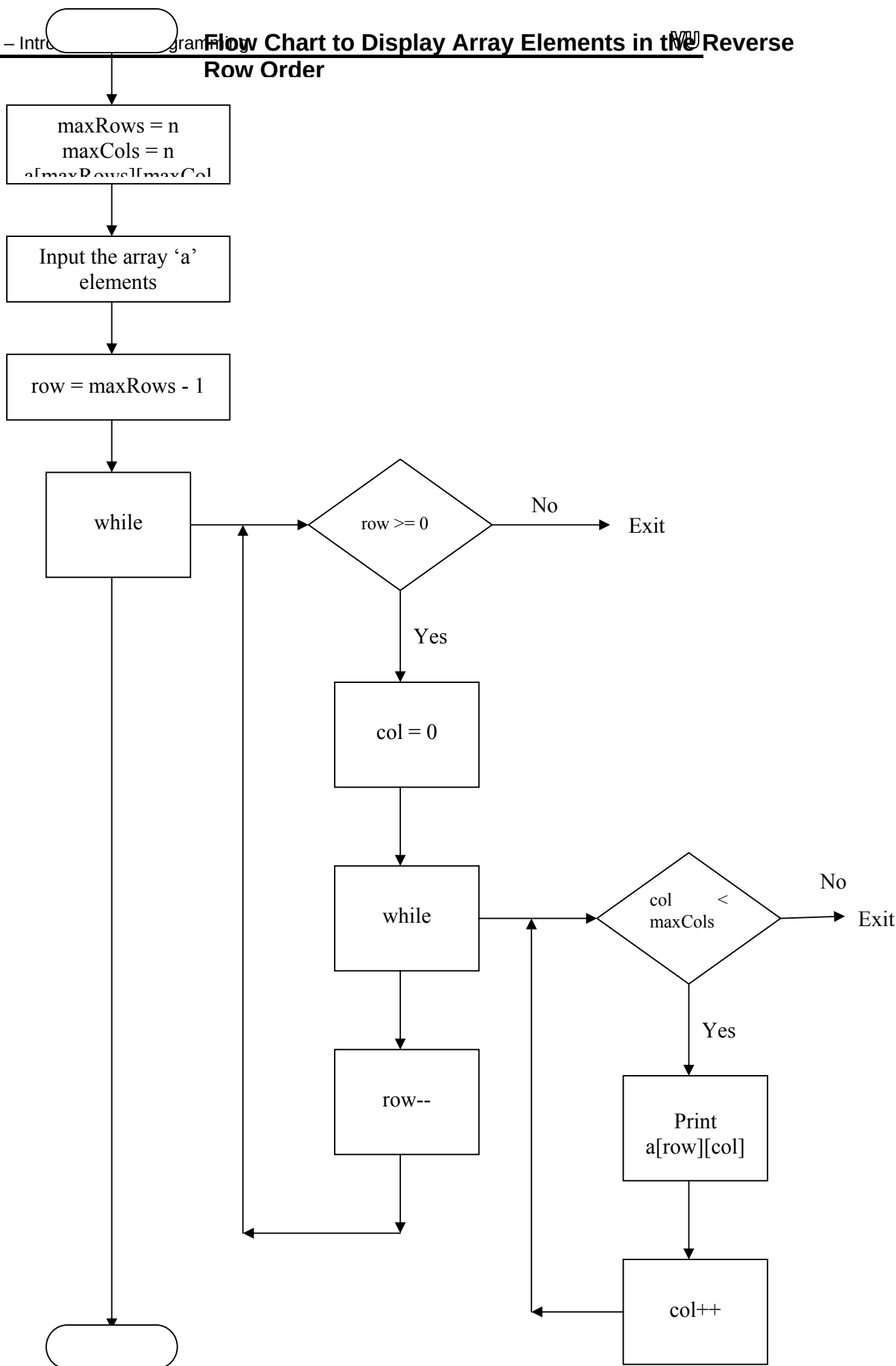
for ( row = maxRows-1; row >= 0; row --)
{
    for ( col = 0; col < maxCols; col ++)
    {
        cout << a [row][col] << '\t';
    }
    cout << '\n';
}
```

Note the '\t' character in the above code. It is a tab character that displays tab (spaces) at the cursor position on the screen. Similarly '\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.



```
/* 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])
```

```
{
    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';
    }
}
```

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.

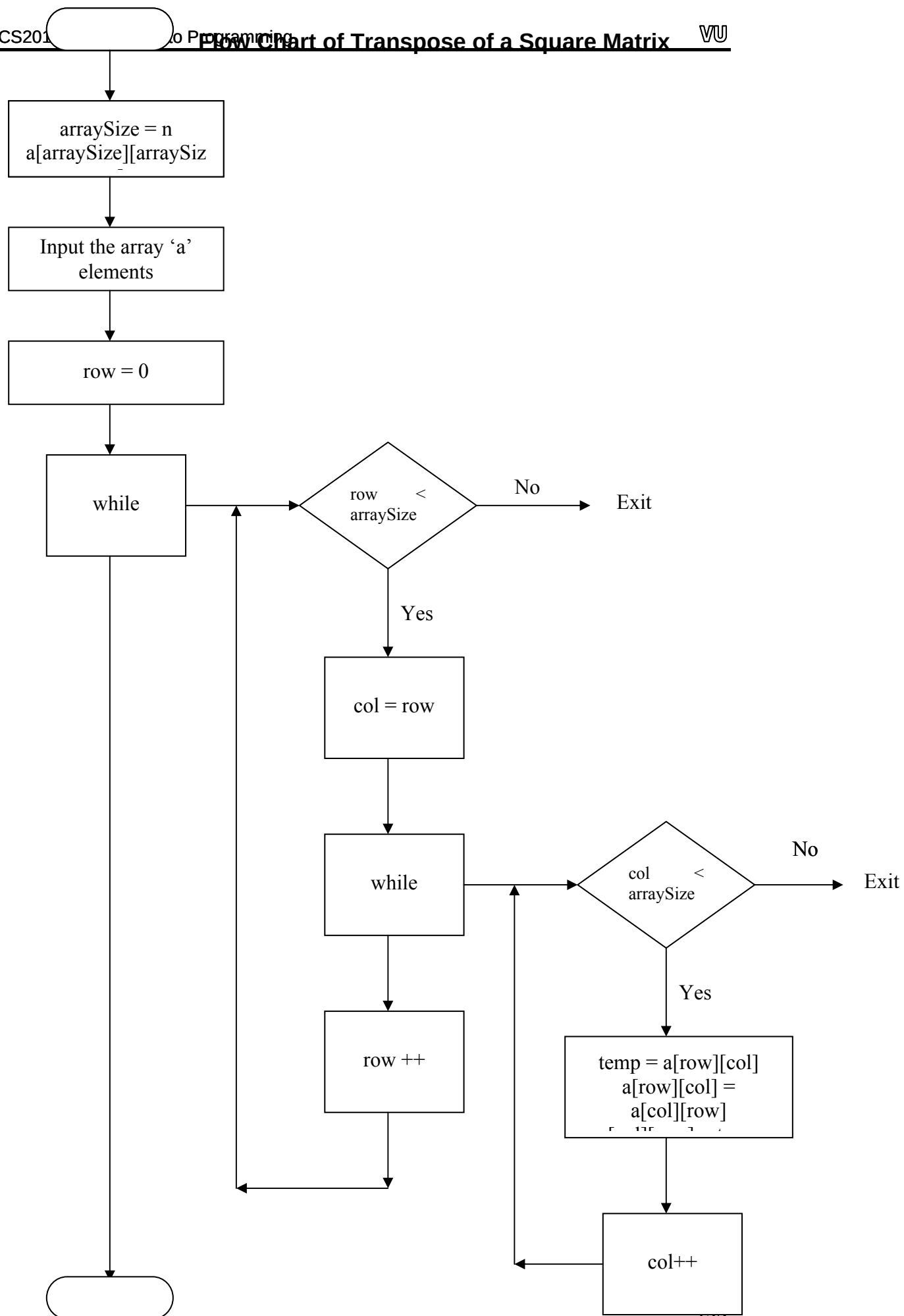
We write a pair of nested loops:

```
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.



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";
```

```
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];
```

```
        a[col][row] = temp;    //Take out the original value
    }
}
```

## 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.

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

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'

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 `getInput()` 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 */
```

```
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)
```

```

    {
        /* 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;
        }
    }
}

```

```
    }  
}  
  
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.

## 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

### 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.

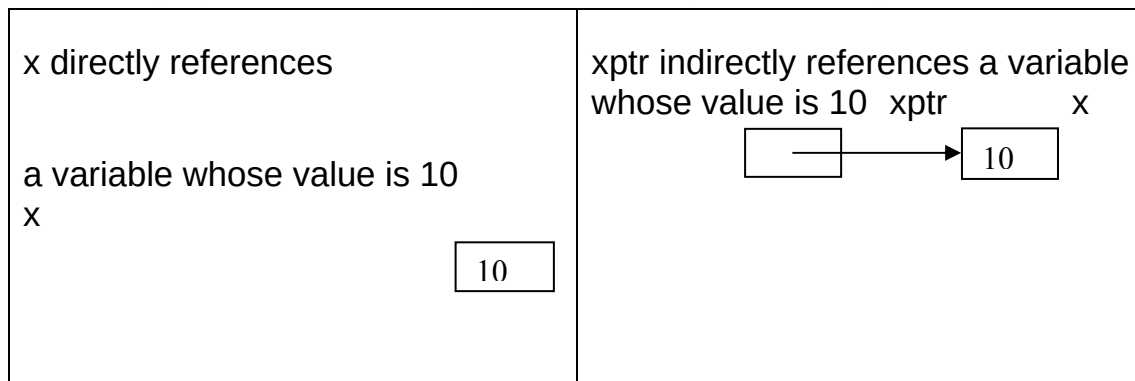
Pointers are a special type of variables in which a memory address is stored. They contain a memory address, not the value of the variable. The concept of the pointers can be well understood from the following example.

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

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.



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

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:

```
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 ;
```

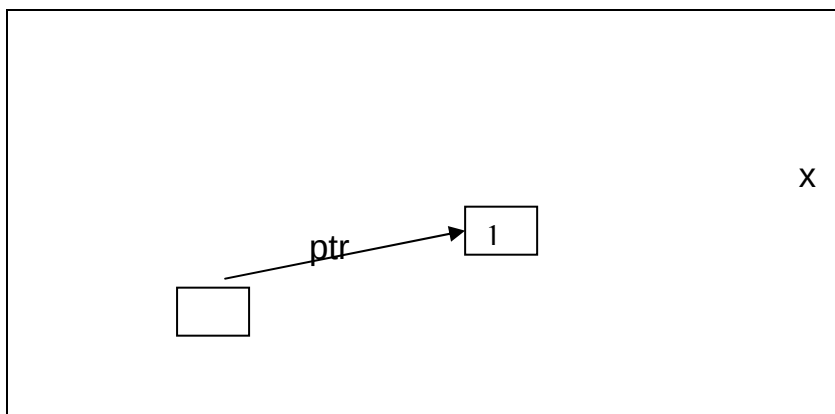
```
int x ;
```

```
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 ;
```

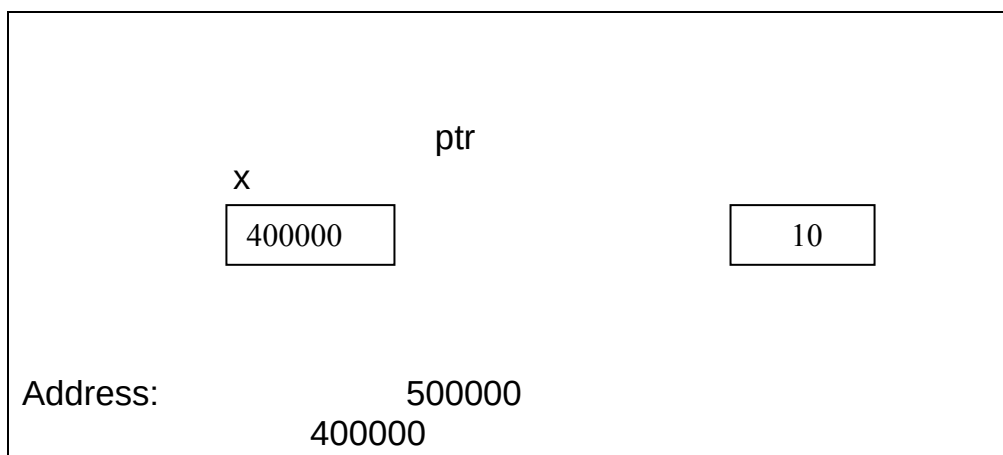
This statement assigns the memory address of the location *x* to the pointer *ptr*. The following figure shows a schematic representation of memory after the preceding assignment is executed.



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.

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.



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:

```
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*<sup>position in a file</sup> 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.

```
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      yptr = &y ;           // address of y is stored in  
xptr      xptr = &x ;           // address of x is stored in  
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)  
{  
    ... ..  
}
```

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 ++)
        {
```

```
        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;
    }
}
```

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 |   |   |   |   |   |

### 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 within 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

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 ;
```

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.

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.

We write a function `convertToUpper`, 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 convertToUpper (char *)
main ()
{
    char s [30] = "Welcome To Virtual University" ;
    cout << "The string before conversion is: " << s << endl ;
    convertToUpper ( s ) ;    //function call
    cout << "The string after conversion is: " << s ;
}

void convertToUpper (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.

2. Write a program, which converts a string of uppercase letters into its corresponding lowercase letters string.

## 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

### *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

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;
}
```

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, `yptr` is pointing to the 2<sup>nd</sup> 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 2<sup>nd</sup> or 3<sup>rd</sup> 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, `yptr` 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 2<sup>nd</sup> 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 */
```

```
#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 6<sup>th</sup> 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 6<sup>th</sup> 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 ++ ;
    }
}
```

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;
}
```

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,

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:

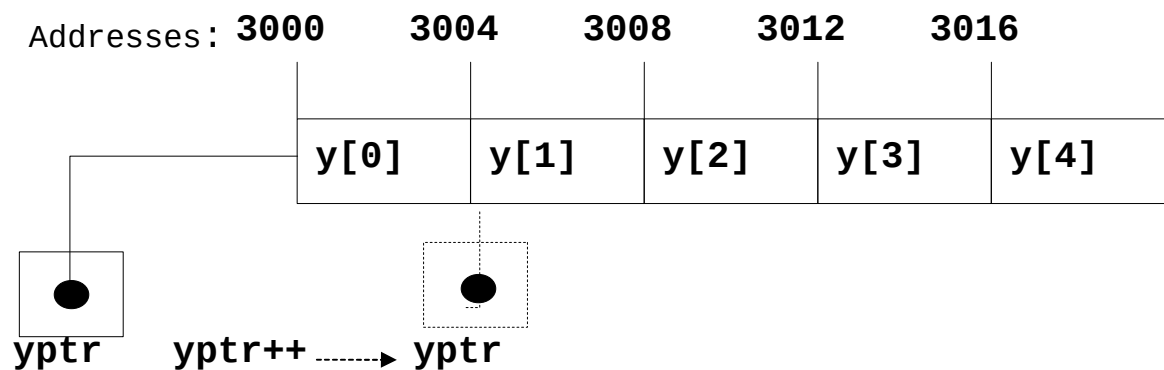
```
/* Program using the pointer subtraction */  
  
#include <iostream.h>  
  
main ()  
{  
    int y[10], *yptr1, *yptr2;  
  
    yptr1 = &y[0];  
    yptr2 = &y[3];  
  
    cout << " The difference = " << yptr2 - yptr1;  
}
```

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 4<sup>th</sup> 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.



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 ;
```

```
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
```

## 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';
```

```
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 2<sup>nd</sup> element of the array *strA* and *ptrB* is pointing to the 2<sup>nd</sup> element of the array *strB*. In the 2<sup>nd</sup> repetition, the loop condition will be tested. If *ptrA* is not pointing to a null character the assignment for the 2<sup>nd</sup> 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.

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

- 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

## 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

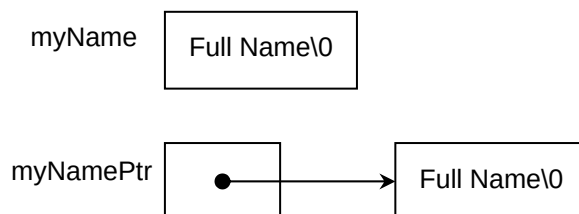
### Pointers (continued)

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.

Diagram illustrating a 1D array structure with 12 elements. The array is represented as a horizontal row of boxes. The indices above the boxes are [0], [1], [2], [3], [4], [5], [6], [7], [8], [9], [0], [1]. The values inside the boxes are 75, 72, 68, 82, 80, 79, 69, 67, 73, 77, 83, 80. Arrows indicate the mapping of 2D coordinates to 1D indices: the 1st row 1st col points to the first element (75), and the 2nd row 1st col points to the 11th element (83).

## 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]'.

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
```

```

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,

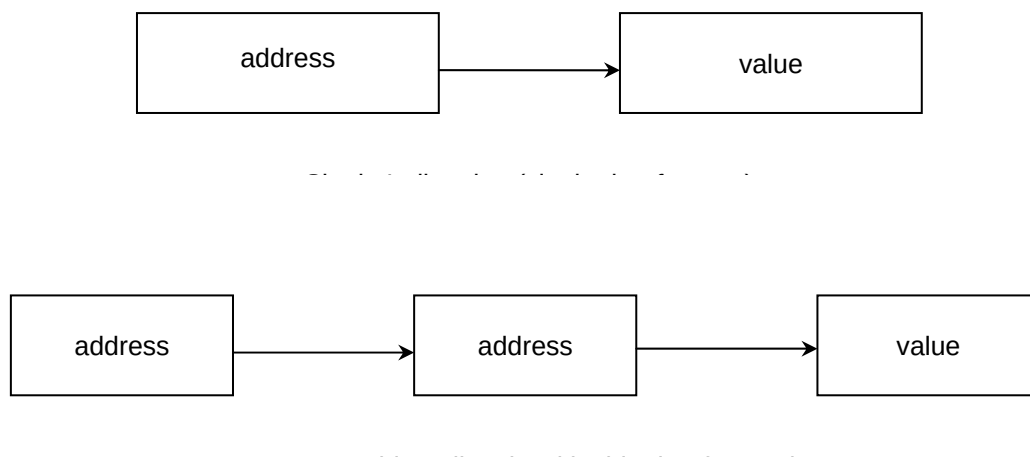
the function needs to know the number of columns so that it has much information about the end and start of rows within an array.

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

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.

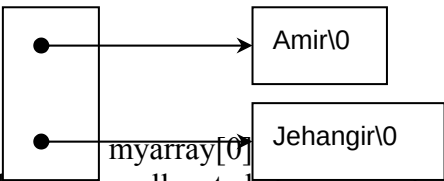
If we use the conventional two-dimensional array like a [5] [10], there is no way of using variable space for rows. All the rows will have fixed ‘10’ number of columns in this case. But in case of an Array of Pointers, we can allocate variable space. An array of pointers is used to store pointers in 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"};
```

myarray



For first pointer myarray[0], 9 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.

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
```

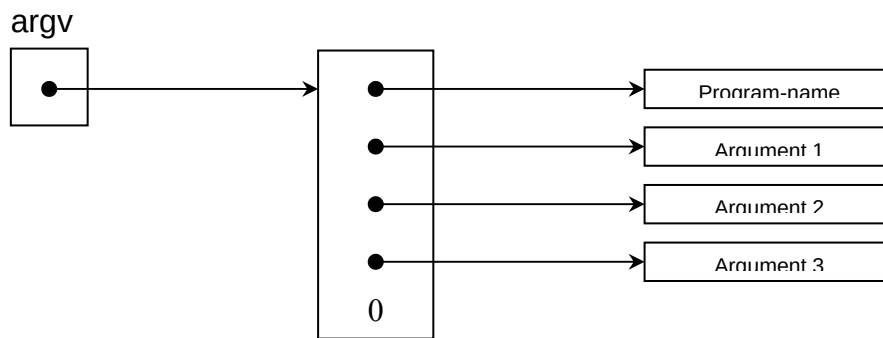
We have so far been taking input using the ‘cout’ and ‘cin’ in the program. But now we can also pass arguments from the command line just before executing the program. For this purpose, we will need a mechanism. In C, this can be done by using ‘argc’ and ‘argv’ arguments inside the main() function as:

```
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.



Now the command line arguments can be accessed from inside the program using ‘argc’ and ‘argv’ variables. It will be an interesting experience for you to try out the following code:

```

/* Accessing the command line arguments */
#include <iostream.h>

main(int argc, char **argv)
{
    cout << argc << endl;
    cout << *argv;
}

```

If we run this program without any argument, then what should be the answer. It will be not correct to think that the argc (number of arguments) is zero as we have not passed any argument. It counts program name as the first argument. So programs written in C/C++ know their names supplied in the first command-line argument. By running the above program, we can have the following output:

```

c:\dev-cpp\work>program
1
program

```

Here we see that the number of arguments is 1 with the first argument as the program name itself. You have to go to the command prompt to provide the command line arguments or you can discuss on the discussion board, how to use Dev-C++ to pass command line arguments.

The command line arguments are separated by spaces. You can provide command line arguments to a program as under:

```

c:\dev-cpp\work>program 1 2

```

Here the number of arguments (argc) will be 3. The argument “1” and “2” are available inside the program as character strings. Therefore, you have to convert them into integers to ensure their usage as numbers.

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;
}
```

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

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?

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

'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;
```

```

        } 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

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

## 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.

## 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)

### 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 continuous. 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, **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                                                                                                                                          |
|------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>int isdigit( int c )</b>  | Returns <b>true</b> if <b>c</b> is a digit and <b>false</b> otherwise.                                                                               |
| <b>int isalpha( int c )</b>  | Returns <b>true</b> if <b>c</b> is a letter and <b>false</b> otherwise.                                                                              |
| <b>int isalnum( int c )</b>  | Returns <b>true</b> if <b>c</b> is a digit or a letter and <b>false</b> otherwise.                                                                   |
| <b>int isxdigit( int c )</b> | Returns <b>true</b> if <b>c</b> is a hexadecimal digit character and <b>false</b> otherwise.                                                         |
| <b>int islower( int c )</b>  | Returns <b>true</b> if <b>c</b> is a lowercase letter and <b>false</b> otherwise.                                                                    |
| <b>int isupper( int c )</b>  | Returns <b>true</b> if <b>c</b> is an uppercase letter; <b>false</b> otherwise.                                                                      |
| <b>int tolower( int c )</b>  | If <b>c</b> is an uppercase letter, <b>tolower</b> returns <b>c</b> as a lowercase letter. Otherwise, <b>tolower</b> returns the argument unchanged. |
| <b>int toupper( int c )</b>  | If <b>c</b> is a lowercase letter, <b>toupper</b> returns <b>c</b> as an uppercase letter. Otherwise, <b>toupper</b> returns the argument unchanged. |

|                             |                                                                                                                                                                                                            |
|-----------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>int isspace( int c )</b> | Returns <b>true</b> if <b>c</b> is a white-space character—newline ('\n'), space (' '), form feed ('\f'), carriage return ('\r'), horizontal tab ('\t'), or vertical tab ('\v')—and <b>false</b> otherwise |
| <b>int iscntrl( int c )</b> | Returns <b>true</b> if <b>c</b> is a control character and <b>false</b> otherwise.                                                                                                                         |
| <b>int ispunct( int c )</b> | Returns <b>true</b> if <b>c</b> is a printing character other than a space, a digit, or a letter and <b>false</b> otherwise.                                                                               |
| <b>int isprint( int c )</b> | Returns <b>true</b> value if <b>c</b> is a printing character including space (' ') and <b>false</b> otherwise.                                                                                            |
| <b>int isgraph( int c )</b> | Returns <b>true</b> if <b>c</b> is a printing character other than space (' ') and <b>false</b> 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>
```

```
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

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                                               |
|---------------------------------------------------------------------------|-----------------------------------------------------------|
| <b>double atof( const char *nPtr )</b>                                    | Converts the string <b>nPtr</b> to <b>double</b> .        |
| <b>int atoi( const char *nPtr )</b>                                       | Converts the string <b>nPtr</b> to <b>int</b> .           |
| <b>long atol( const char *nPtr )</b>                                      | Converts the string <b>nPtr</b> to long <b>int</b> .      |
| <b>double strtod( const char *nPtr, char **endPtr )</b>                   | Converts the string <b>nPtr</b> to <b>double</b> .        |
| <b>long strtol( const char *nPtr, char **endPtr, int base )</b>           | Converts the string <b>nPtr</b> to <b>long</b> .          |
| <b>unsigned long strtoul( const char *nPtr, char **endPtr, int base )</b> | Converts the string <b>nPtr</b> to <b>unsigned long</b> . |

#### 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

```

```

        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                                                                                                                                                                                       |
|----------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>char *strcpy( char *s1, const char *s2 )</b>                | Copies string <b>s2</b> into character array <b>s1</b> . The value of <b>s1</b> is returned.                                                                                                               |
| <b>char *strncpy( char *s1, const char *s2, size_t n )</b>     | Copies at most <b>n</b> characters of string <b>s2</b> into array <b>s1</b> . The value of <b>s1</b> is returned.                                                                                          |
| <b>char *strcat( char *s1, const char *s2 )</b>                | Appends string <b>s2</b> to array <b>s1</b> . The first character of <b>s2</b> overwrites the terminating null character of <b>s1</b> . The value of <b>s1</b> is returned.                                |
| <b>char *strncat( char *s1, const char *s2, size_t n )</b>     | Appends at most <b>n</b> characters of string <b>s2</b> to array <b>s1</b> . The first character of <b>s2</b> overwrites the terminating null character of <b>s1</b> . The value of <b>s1</b> is returned. |
| <b>int strcmp( const char *s1, const char *s2)</b>             | Compares string <b>s1</b> to <b>s2</b> . Returns a negative number if <b>s1 &lt; s2</b> , zero if <b>s1 == s2</b> or a positive number if <b>s1 &gt; s2</b>                                                |
| <b>int strncmp( const char *s1, const char *s2, size_t n )</b> | Compares up to <b>n</b> characters of string <b>s1</b> to <b>s2</b> . Returns a negative number if <b>s1 &lt; s2</b> , zero if <b>s1 == s2</b> or a positive number if <b>s1 &gt; s2</b> .                 |
| <b>int strlen ( const char *s)</b>                             | Determines the length of string <b>s</b> . 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
```

```

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

```

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                                                                                                                                                                                                                             |
|----------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>char *strchr( const char *s, int c );</b>             | Locates the first occurrence of character <b>c</b> in string <b>s</b> . If <b>c</b> is found, a pointer to <b>c</b> in <b>s</b> is returned. Otherwise, a <b>NULL</b> pointer is returned.                                                       |
| <b>size_t strcspn( const char *s1, const char *s2 );</b> | Determines and returns the length of the initial segment of string <b>s1</b> consisting of characters not contained in string <b>s2</b> .                                                                                                        |
| <b>size_t strspn( const char *s1, const char *s2 );</b>  | Determines and returns the length of the initial segment of string <b>s1</b> consisting only of characters contained in string <b>s2</b> .                                                                                                       |
| <b>char *strpbrk( const char *s1, const char *s2 );</b>  | Locates the first occurrence in string <b>s1</b> of any character in string <b>s2</b> . If a character from string <b>s2</b> is found, a pointer to the character in string <b>s1</b> is returned. Otherwise, a <b>NULL</b> pointer is returned. |
| <b>char *strrchr( const char *s, int c );</b>            | Locates the last occurrence of <b>c</b> in string <b>s</b> . If <b>c</b> is found, a pointer to <b>c</b> in string <b>s</b> is returned.                                                                                                         |

|                                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|--------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                        | Otherwise, a <b>NULL</b> pointer is returned.                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>char *strstr( const char *s1, const char *s2 );</b> | Locates the first occurrence in string <b>s1</b> of string <b>s2</b> . If the string is found, a pointer to the string in <b>s1</b> is returned. Otherwise, a <b>NULL</b> pointer is returned.                                                                                                                                                                                                                                                                                               |
| <b>char *strtok( char *s1, const char *s2 );</b>       | A sequence of calls to <b>strtok</b> breaks string <b>s1</b> into “tokens”—logical pieces such as words in a line of text—separated by characters contained in string <b>s2</b> . The first call contains <b>s1</b> as the first argument, and subsequent calls to continue tokenizing the same string contain <b>NULL</b> 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, <b>NULL</b> 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");
```

```
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
```

## 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  
14.3, 14.4, 14.5, 14.6

Chapter 14

### **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

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
```

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 understand 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,

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`, 2<sup>nd</sup> in `c2` and 3<sup>rd</sup> 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, 2<sup>nd</sup> word is salary etc, so that we can read the first word in a *string* and 2<sup>nd</sup> 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  |

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

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 2<sup>nd</sup> 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:

| 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

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 to 2Ghz. When we talk about hard disk, we say its average access time is 7 mili sec. It

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
 */
```

```
*/  
  
#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 successfully 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();  
}
```

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;
```

```
        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 output 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.

Open a file explicitly with open function

Always apply the error checking mechanism while handling with files.

## 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

### *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.

## 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

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");

```

```
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.

## 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 beginning of the file.

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

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();
```

```
}
```

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

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 **Rawalpindi** 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 23<sup>rd</sup> 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.

- 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 23<sup>rd</sup>'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 23<sup>rd</sup>'s student starts from 2201<sup>st</sup> byte and goes to 2300<sup>th</sup> 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?

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 100<sup>th</sup> location, you have to read in sequence one by one. But if you use **seekg ()** function to go to 100<sup>th</sup> 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

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.

### 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
}
```

## Sample Program 2

/\* This sample code demonstrates 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 backward
```

```
    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:

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.

## 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

### 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

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 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 sukhra, rawalpindi, etc and country like Pakistan. It can be written in C language as:

```

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

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;
```

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.

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;
```

```
// 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

## 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 2<sup>nd</sup> 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

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 ;
    }
}
```

```

        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
    {

```

```
    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

## 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 ( ) ;
}
```

```

/* 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

```

|                                                           |
|-----------------------------------------------------------|
| Jamil<br>CS201<br>31<br>4<br>Faisal<br>CS201<br>25<br>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 ASCII 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 2<sup>nd</sup> 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:

```
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.