1

Algorithms, Flowcharts & Program Design

Unit Structure:

1.1 OBJECTIVES

After completing this chapter, you will be able to:

- Understand the basics and usefulness of an algorithm,
- Analyse various algorithms,
- Understand a flowchart and its advantages and limitations,
- Steps involved in designing a program.

1.2 INTRODUCTION

A computer is a useful tool for solving a great variety of problems. To make a computer do anything (i.e. solve a problem), you have to write a *computer program*. In a computer program you tell a computer, step by step, exactly what you want it to do. The computer then executes the program, following each step mechanically, to accomplish the end goal.

The sequence of steps to be performed in order to solve a problem by the computer is known as an ***algorithm***.

Flowchart is a graphical or symbolic representation of an algorithm. It is the diagrammatic representation of the step-by-step solution to a given problem.

Program Design consists of the steps a programmer should do before they start coding the program in a specific language. Proper program design helps other programmers to maintain the program in the future.

1.3 ALGORITHMS

Look around you. Computers and networks are everywhere, enabling an intricate web of complex human activities: education, commerce, entertainment, research, manufacturing, health management, communication, even war. Of the two main technological underpinnings of this amazing proliferation, one is the breathtaking pace with which advances in microelectronics and chip design have been bringing us faster and faster hardware. The other is efficient algorithms that are fuelling the computer revolution.

In mathematics, computer science, and related subjects, an *algorithm* is a finite sequence of steps expressed for solving a problem. An *algorithm* can be defined as "a process that performs some sequence of operations in order to solve a given problem". Algorithms are used for calculation, data processing, and many other fields.

In computing, algorithms are essential because they serve as the systematic procedures that computers require. A good algorithm is like using the right tool in the workshop. It does the job with the right amount of effort. Using the wrong algorithm or one that is not clearly defined is like trying to cut a piece of plywood with a pair of scissors: although the job may get done, you have to wonder how effective you were in completing it.

Let us follow an example to help us understand the concept of algorithm in a better way. Let's say that you have a friend arriving at the railway station, and your friend needs to get from the railway station to your house. Here are three different ways (algorithms) that you might give your friend for getting to your home.

- **The taxi/auto-rickshaw algorithm:**
  - Go to the taxi/auto-rickshaw stand.
  - Get in a taxi/auto-rickshaw.
  - Give the driver my address.
- **The call-me algorithm:**
  - When your train arrives, call my mobile phone.

- Meet me outside the railway station.

❖ **The bus algorithm:**

- Outside the railway station, catch bus number 321.
- Transfer to bus 308 near Kurla station.
- Get off near Kalina University.
- Walk two blocks west to my house.

All these three algorithms accomplish the same goal, but each algorithm does it in a different way. Each algorithm also has a different cost and a different travel time. Taking a taxi/auto-rickshaw, for example, is the fastest way, but also the most expensive. Taking the bus is definitely less expensive, but a whole lot slower. You choose the algorithm based on the circumstances.

In computer programming, there are often many different algorithms to accomplish any given task. Each algorithm has advantages and disadvantages in different situations. Sorting is one place where a lot of research has been done, because computers spend a lot of time sorting lists.

Three reasons for using algorithms are *efficiency, abstraction* and *reusability.*

❖ **Efficiency**: Certain types of problems, like sorting, occur often in computing. Efficient algorithms must be used to solve such problems considering the time and cost factor involved in each algorithm.

❖ **Abstraction**: Algorithms provide a level of abstraction in solving problems because many seemingly complicated problems can be distilled into simpler ones for which well-known algorithms exist. Once we see a more complicated problem in a simpler light, we can think of the simpler problem as just an abstraction of the more complicated one. For example, imagine trying to find the shortest way to route a packet between two gateways in an internet. Once we realize that this problem is just a variation of the more general shortest path problem, we can solve it using the generalised approach.

❖ **Reusability**: Algorithms are often reusable in many different situations. Since many well-known algorithms are the generalizations of more complicated ones, and since many complicated problems can be distilled into simpler ones, an efficient means of solving certain simpler problems potentially lets us solve many complicated problems.

1.3.1 Expressing Algorithms:

Algorithms can be expressed in many different notations, including *natural languages*, *pseudocode*, *flowcharts* and *programming languages*. *Natural language* expressions of algorithms tend to be verbose and ambiguous, and are rarely used for complex or

technical algorithms. *Pseudocode* and *flowcharts* are structured ways to express algorithms that avoid many ambiguities common in natural language statements, while remaining independent of a particular implementation language. *Programming languages* are primarily intended for expressing algorithms in a form that can be executed by a computer, but are often used to define or document algorithms.

Sometimes it is helpful in the description of an algorithm to supplement small flowcharts with natural language and/or arithmetic expressions written inside block diagrams to summarize what the flowcharts are accomplishing.

Consider an example for finding the largest number in an unsorted list of numbers.

The solution for this problem requires looking at every number in the list, but only once at each.

1) *Algorithm using natural language statements*:
   a) Assume the first item is largest.
   b) Look at each of the remaining items in the list and if it is larger than the largest item so far, make a note of it.
   c) The last noted item is the largest item in the list when the process is complete.

2) Algorithm using pseudocode:
largest = L_0
for each *item* in the list (*Length(L)* ≥ *1*), **do**
    if the *item* ≥ largest, **then**
        largest = the *item*
return *largest*

1.3.2 Benefits of using Algorithms:

The use of algorithms provides a number of benefits. One of these benefits is in the *development of the procedure* itself, which involves identification of the processes, major decision points, and variables necessary to solve the problem. Developing an algorithm allows and even forces examination of the solution process in a rational manner. Identification of the processes and decision points reduces the task into a series of smaller steps of more manageable size. Problems that would be difficult or impossible to solve in entirety can be approached as a series of small, solvable sub-problems.

By using an algorithm, decision making becomes a more *rational process*. In addition to making the process more rational, use of algorithm will make the process more efficient and more consistent. *Efficiency* is an inherent result of the analysis and specification process. *Consistency* comes from both the use of the same specified process and increased skill in applying the process. An algorithm serves as a mnemonic device and helps ensure that variables or parts of

the problem are not ignored. Presenting the solution process as an algorithm allows more precise communication. Finally, separation of the procedure steps facilitates division of labour and development of expertise.

A final benefit of the use of an algorithm comes from the *improvement* it makes possible. If the problem solver does not know what was done, he or she will not know what was done wrong. As time goes by and results are compared with goals, the existence of a specified solution process allows identification of weaknesses and errors in the process. Reduction of a task to a specified set of steps or algorithm is an important part of analysis, control and evaluation.

1.3.3 General Approaches in Algorithm Design:

In a broad sense, many algorithms approach problems in the same way. Thus, it is often convenient to classify them based on the approach they employ. One reason to classify algorithms is to gain an insight about an algorithm and understand its general approach. This can also give us ideas about how to look at similar problems for which we do not know algorithms. Of course, some algorithms defy classification, whereas others are based on a combination of approaches. This section presents some common approaches.

1.3.3.1 Randomized Algorithms:

Randomized algorithms rely on the statistical properties of random numbers. One example of a randomized algorithm is quicksort.

Imagine an unsorted pile of cancelled checks by hand. In order to sort this pile we place the checks numbered less than or equal to what we may think is the median value in one pile, and in the other pile we place the checks numbered greater than this. Once we have the two piles, we divide each of them in the same manner and repeat the process until we end up with one check in every pile. At this point the checks are sorted.

1.3.3.2 Divide-and-conquer Algorithms:

Divide-and-conquer algorithms revolve around 3 steps: divide, conquer and combine. In the divide step, we divide the data into smaller, more manageable pieces. In the conquer step, we process each division by performing some operations on it. In the combine step, we recombine the processed divisions. An example of the divide-and-conquer algorithm is merge sort.

As before, imagine an unsorted pile of cancelled checks by hand. We begin by dividing the pile in half. Next, we divide each of the resulting two piles in half and continue this process until we end up with one check in every pile. Once all piles contain a single check, we merge the piles two by two so that each pile is a sorted combination of the two that were merged. Merging continues until we end up with one big pile again, at which point the checks are sorted.

1.3.3.3 Dynamic-programming solutions:

Dynamic-programming solutions are similar to divide-and-conquer methods in that both solve problems by breaking larger problems into sub-problems whose results are later recombined. However, the approaches differ in how sub-problems are related. In divide-and-conquer algorithms, each sub-problem is independent of the others. Therefore we solve each sub-problem using recursion and combine its results with the results of other sub-problems. In dynamic-programming solutions, sub-problems are not independent of one another. A dynamic-programming solution is better than a divide-and-conquer approach because the latter approach will do more work than necessary, as shared sub-problems are solved more than once. However, if the sub-problems are independent and there is no repetition, using divide-and-conquer algorithms is much better than using dynamic-programming.

An example of dynamic-programming is finding the shortest path to reach a point from a vertex in a weighted graph.

1.3.3.4 Greedy Algorithms:

Greedy algorithms make decisions that look best at the moment. In other words, they make decisions that are locally optimal in the hope that they will lead to globally optimal solutions. The greedy method extends the solution with the best possible decision at an algorithmic stage based on the current local optimum and the best decision made in a previous stage. It is not exhaustive, and does not give accurate answer to many problems. But when it works, it will be the fastest method.

One example of a greedy algorithm is Huffman coding, which is an algorithm for data compression.

1.3.3.5 Approximation Algorithms:

Approximation algorithms are algorithms that do not compute optimal solutions; instead, they compute solutions that are “good enough”. Often we use approximation algorithms to solve problems that are computationally expensive but are too significant to give up altogether. The travelling salesman problem is one example of a problem usually solved using an approximation algorithm.

1.3.4 Analysis of Algorithms:

Whether we are designing an algorithm or applying one that is widely accepted, it is important to understand how the algorithm will perform. There are a number of ways we can look at an algorithm’s performance, but usually the aspect of most interest is how fast the algorithm will run. In some cases, if an algorithm uses significant storage, we may be interested in its space requirement as well.

Analysis of algorithms is the theoretical study of computer program performance and resource usage, and is often practised abstractly without the use of specific programming language or implementation. The practical goal of algorithm analysis is to predict the performance of different algorithms in order to guide program design decisions. There are many reasons to understand the performance of an algorithm. For example, we often have a choice of several algorithms when solving problems (like sorting). Understanding how each performs helps us differentiate between them. Understanding the burden an algorithm places on an application also helps us plan how to use the algorithm more effectively. For instance, garbage collection algorithms, algorithms that collect dynamically allocated storage to return to the heap, require considerable time to run. Knowing this we can be careful to run them only at opportune moments, just as Java does, for example.

Algorithm analysis is important in practice because the accidental or unintentional use of an inefficient algorithm can significantly impact system performance. In time-sensitive applications, an algorithm taking too long to run can render its results outdated or useless. An inefficient algorithm can also end up requiring an uneconomical amount of computing power or storage in order to run, again rendering it practically useless.

1.3.4.1 Worst-Case Analysis:

Most algorithms do not perform the same in all cases; normally an algorithm's performance varies with the data passed to it. Typically, three cases are recognized: *the best case, average case and worst case*. For any algorithm, understanding what constitutes each of these cases is an important part of analysis because performance can vary significantly between them.

Consider a simple algorithm such as linear search. Linear search is a natural but inefficient search technique in which we look for an element simply by traversing a set from one end to the other. In the best case, the element we are looking for is the first element we inspect, so we end up traversing only a single element. In the worst case, however, the desired element is the last element that we inspect, in which we end up traversing all the elements. On average, we can expect to find the element somewhere in the middle.

A basic understanding of how an algorithm performs in all cases is important, but usually we are most interested in how an algorithm performs in the worst case. There are four reasons why algorithms are generally analyzed by their worst case:

- Many algorithms perform to their worst case a large part of the time. For example, the worst case in searching occurs when we do not find what we are looking for at all. This frequently happens in database applications.
- The best case is not very informative because many algorithms perform exactly the same in the best case. For example, nearly

all searching algorithms can locate an element in one inspection at best, so analyzing this case does not tell us much.

- Determining average-case performance is not always easy. Often it is difficult to determine exactly what the "average case" even is; since we can seldom guarantee precisely how an algorithm will be exercised.
- The worst case gives us an upper bound on performance. Analyzing an algorithm's worst case guarantees that it will never perform worse than what we determine. Therefore, we know that the other cases must perform at least better than the worst case.

Worst case analysis of algorithms is considered to be crucial to applications such as games, finance and robotics. Although worst case analysis is the metric for many algorithms, it is worth noting that there are exceptions. Sometimes special circumstances let us base performance on the average case (for example quick-sort algorithm).

1.3.4.2 O-notation:

O-notation, also known as *Big O-notation*, is the most common notation used to express an algorithm's performance in a formal manner. Formally, O-notation expresses the upper bound is a function within a constant factor. Specifically, if g(n) is an upper bound of f(n), then for some constant c it is possible to find the value of n, call it n_0, for which any value of $n \geq n_0$ will result in $f(n) \leq cg(n)$.

Normally we express an algorithm's performance as a function of the size of the data it processes. That is, for some data of size n, we describe its performance with some function f(n). Primarily we are interested only in the growth rate of f, which describes how quickly the algorithm's performance will degrade as the size of data it processes becomes arbitrarily large. An algorithm's growth rate, or order of growth, is significant because ultimately it describes how efficient the algorithm is for aritrary inputs. O-notation reflects an algorithm's order of growth.

Simple Rules for O-notation:

O-notation lets us focus on the big picture. When faced with a complicated function like $3n^2 + 4n + 5$, we just replace it with O(f(n)), where f(n) is as simple as possible. In this particular example we'd use $O(n^2)$, because the quadratic portion of the sum dominates the rest of the function. Here are some rules that help simplify functions by omitting dominated terms:

1. We can ignore constant terms because as the value of n becomes larger and larger, eventually constant terms will become insignificant. For example, if T(n) = n+50 describes the running time of an algorithm, and n, the size of data it processes, is only 1024, the constant term in this expression constitutes less than 5% of the running time.

2. We can ignore multiplicative constants because they too will become insignificant as the value of n increases. For example, $14n^2$ becomes n^2.
3. We need to consider the highest-order term because as n increases higher-order terms quickly outweigh the lower-order terms. That is, n^a dominates n^b if a > b. For instance, n^2 dominates n.
4. We also must note that an exponential term dominates any polynomial term in the expression. For example, 3^n dominates n^5 (it even dominates 2^n).
5. Similarly, a polynomial term dominates any logarithmic term used in the expression. For example, n dominates $(\log n)^3$. This also means, for example, that n^2 dominates n log n.

1.3.4.3 O-Notation Example:

This section discusses how these rules help us in predicting an algorithm's performance. Let's look at a specific example demonstrating why they work so well in describing a function's growth rate. Suppose we have an algorithm whose running time is described by the function $T(n) = 3n^2 + 10n + 10$.

Using the rules of O-notation, this function can be simplified to:

$$O(T(n)) = O(3n^2 + 10n + 10) = O(3n^2) = O(n^2)$$

This indicates that the term n^2 will be the one that accounts for the most of the running time as n grows arbitrarily large. We can verify this quantitatively by computing the percentage of the overall running time that each term accounts for as n increases. For example, when n = 10, we have the following:

Running time for $3n^2$: $3(10)^2 / (3(10)^2 + 10(10) + 10) = 73.2\%$
Running time for 10n: $10(10) / (3(10)^2 + 10(10) + 10) = 24.4\%$
Running time for 10: $10 / (3(10)^2 + 10(10) + 10) = 2.4\%$

Already we see that the n^2 term accounts for the majority of the overall running time. Now consider when n = 100:

Running time for $3n^2$: $3(100)^2 / (3(100)^2 + 10(100) + 10) = 96.7\%$
Running time for 10n: $10(100) / (3(100)^2 + 10(100) + 10) = 3.2\%$
Running time for 10: $10 / (3(100)^2 + 10(100) + 10) = 0.1\%$

Here we see that this term accounts for almost all of the running time, while the significance of the other terms diminishes further. Imagine how much of the running time this term would account for if n was 10^6!

Check Your Progress:

1. Write an algorithm to find out the smallest number in an unsorted list.

2. Given above, merge sort is used to sort an unsorted pile of cancelled checks (section 1.3.3.2). Find out other methods that can be used for sorting.

1.4 FLOWCHARTS

A *Flowchart* is a type of diagram (graphical or symbolic) that represents an algorithm or process. Each step in the process is represented by a different symbol and contains a short description of the process step. The flow chart symbols are linked together with arrows showing the process flow direction. A flowchart typically shows the flow of data in a process, detailing the operations/steps in a pictorial format which is easier to understand than reading it in a textual format.

A *flowchart* describes what operations (and in what sequence) are required to solve a given problem. A flowchart can be likened to the blueprint of a building. As we know a designer draws a blueprint before starting construction on a building. Similarly, a programmer prefers to draw a flowchart prior to writing a computer program. Flowcharts are a pictorial or graphical representation of a process. The purpose of all flow charts is to communicate how a process works or should work without any technical or group specific jargon.

Flowcharts are used in analyzing, designing, documenting or managing a process or program in various fields.

Flowcharts are generally drawn in the early stages of formulating computer solutions. Flowcharts often facilitate communication between programmers and business people. These flowcharts play a vital role in the programming of a problem and are quite helpful in understanding the logic of complicated and lengthy problems. Once the flowchart is drawn, it becomes easy to write the program in any high level language. Often we see how flowcharts are helpful in explaining the program to others. Hence, it is correct to say that a flowchart is a must for the better documentation of a complex program.

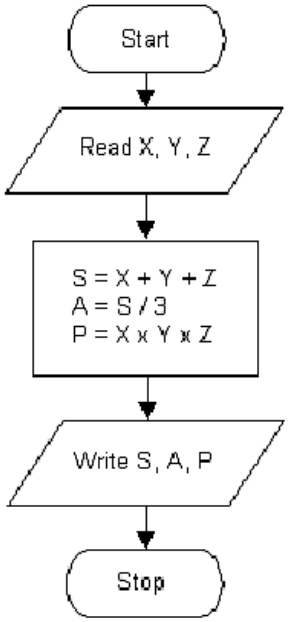
For example, consider that we need to find the sum, average and product of 3 numbers given by the user.

Algorithm for the given problem is as follows:

Read X, Y, Z
Compute Sum (S) as X + Y + Z
Compute Average (A) as S / 3
Compute Product (P) as X x Y x Z

Write (Display) the Sum, Average and Product

Flowchart for the above problem will look like



Now that we have seen an example of a flowchart let us list the advantages and limitations of using flowcharts.

1.4.1 Advantages of Using Flowcharts:

The *benefits of flowcharts* are as follows:

- *Communication*: Flowcharts are better way of communicating the logic of a system to all concerned.
- *Effective analysis*: With the help of flowchart, problem can be analysed in more effective way.
- *Proper documentation*: Program flowcharts serve as a good program documentation, which is needed for various purposes.
- *Efficient Coding*: The flowcharts act as a guide or blueprint during the systems analysis and program development phase.
- *Proper Debugging*: The flowchart helps in debugging process.
- *Efficient Program Maintenance*: The maintenance of operating program becomes easy with the help of flowchart. It helps the programmer to put efforts more efficiently on that part.

1.4.2 Limitations of Using Flowcharts:

Although a flowchart is a very useful tool, there are a few limitations in using flowcharts which are listed below:

- *Complex logic*: Sometimes, the program logic is quite complicated. In that case, flowchart becomes complex and clumsy.
- *Alterations and Modifications*: If alterations are required the flowchart may require re-drawing completely.

- *Reproduction*: As the flowchart symbols cannot be typed, reproduction of flowchart becomes a problem.
- The essentials of what is done can easily be lost in the technical details of how it is done.

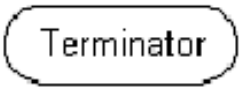
1.4.3 When to Use a Flowchart:

- To communicate to others how a process is done.
- A flowchart is generally used when a new project begins in order to plan for the project.
- A flowchart helps to clarify how things are currently working and how they could be improved. It also assists in finding the key elements of a process, while drawing clear lines between where one process ends and the next one starts.
- Developing a flowchart stimulates communication among participants and establishes a common understanding about the process. Flowcharts also uncover steps that are redundant or misplaced.
- Flowcharts are used to help team members, to identify who provides inputs or resources to whom, to establish important areas for monitoring or data collection, to identify areas for improvement or increased efficiency, and to generate hypotheses about causes.
- It is recommended that flowcharts be created through group discussion, as individuals rarely know the entire process and the communication contributes to improvement.
- Flowcharts are very useful for documenting a process (simple or complex) as it eases the understanding of the process.
- Flowcharts are also very useful to communicate to others how a process is performed and enables understanding of the logic of a process.

1.4.4 Flowchart Symbols & Guidelines:

Flowcharts are usually drawn using some standard symbols; however, some special symbols can also be developed when required. Some standard symbols, which are frequently required for flowcharting many computer programs are shown.

Terminator: An oval flow chart shape indicates the start or end of the process, usually containing the word “Start” or “End”.

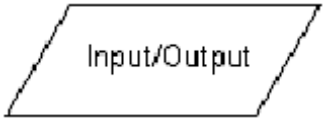


Process: A rectangular flow chart shape indicates a normal/generic process flow step. For example, “Add 1 to X”, “M = M*F” or similar.

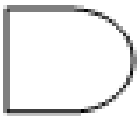
Decision: A diamond flow chart shape indicates a branch in the process flow. This symbol is used when a decision needs to be made, commonly a Yes/No question or True/False test.

Connector: A small, labelled, circular flow chart shape used to indicate a jump in the process flow. Connectors are generally used in complex or multi-sheet diagrams.

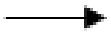
Data: A parallelogram that indicates data input or output (I/O) for a process. Examples: Get X from the user, Display X.



Delay: used to indicate a delay or wait in the process for input from some other process.



Arrow: used to show the flow of control in a process. An arrow coming from one symbol and ending at another symbol represents that control passes to the symbol the arrow points to.

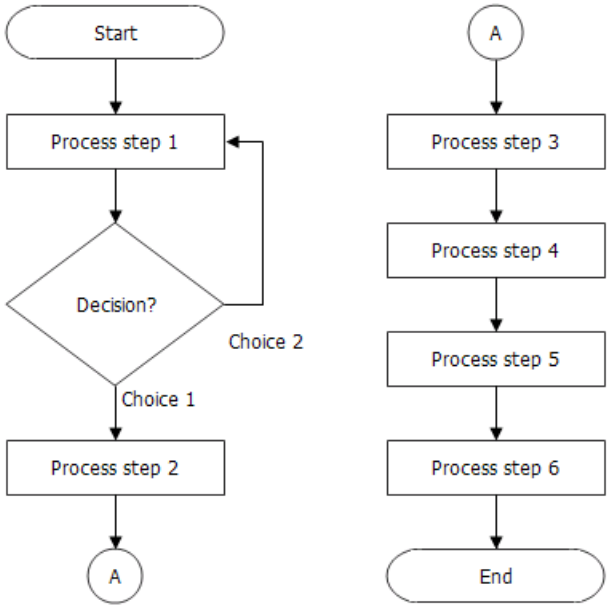


These are the basic symbols used generally. Now, the ***basic guidelines*** for drawing a flowchart with the above symbols are that:

- In drawing a proper flowchart, all necessary requirements should be listed out in logical order.
- The flowchart should be neat, clear and easy to follow. There should not be any room for ambiguity in understanding the flowchart.
- The flowchart is to be read left to right or top to bottom.
- A process symbol can have only one flow line coming out of it.

- For a decision symbol, only one flow line can enter it, but multiple lines can leave it to denote possible answers.
- The terminal symbols can only have one flow line in conjunction with them.

Basic Flowchart



Example:
Consider another problem of finding the largest number between A and B

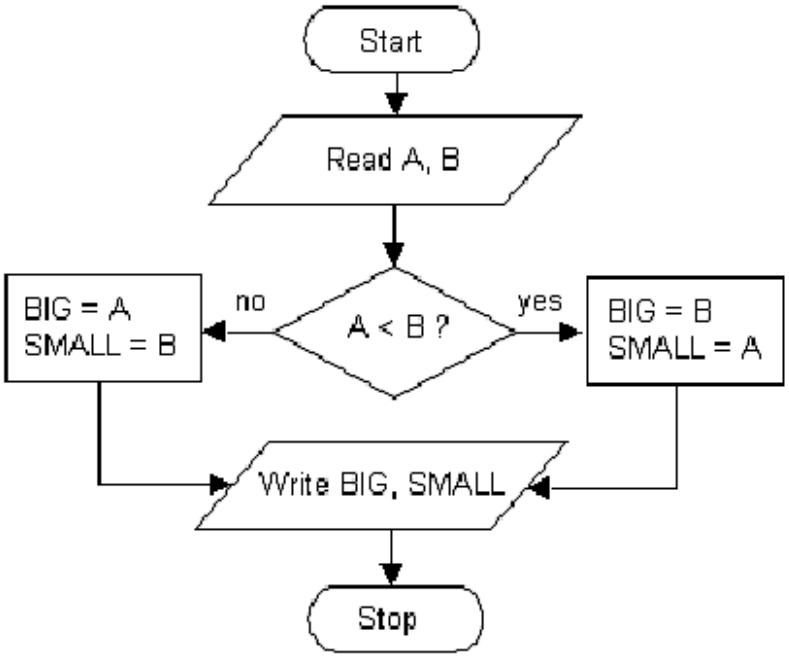
Algorithm for the above problem is as follows:

Read A, B

If A is less than B
    BIG=B
    SMALL = A
Else
    BIG=A
    SMALL = B

Write (Display) BIG, SMALL

Flowchart for the above algorithm will look like:



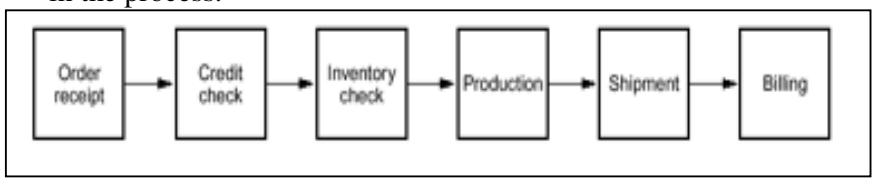
1.4.5 Types of Flowcharts:

- ***High-Level Flowchart:***

A high-level (also called first-level or top-down) flowchart shows the major steps in a process. It illustrates a "birds-eye view" of a process. It can also include the intermediate outputs of each step (the product or service produced), and the sub-steps involved. Such a flowchart offers a basic picture of the process and identifies the changes taking place within the process. It is significantly useful for identifying appropriate team members (those who are involved in the process) and for developing indicators for monitoring the process because of its focus on intermediate outputs.

Most processes can be adequately portrayed in four or five boxes that represent the major steps or activities of the process. In fact, it is a good idea to use only a few boxes, because doing so forces one to consider the most important steps. Other steps are usually sub-steps of the more important ones.

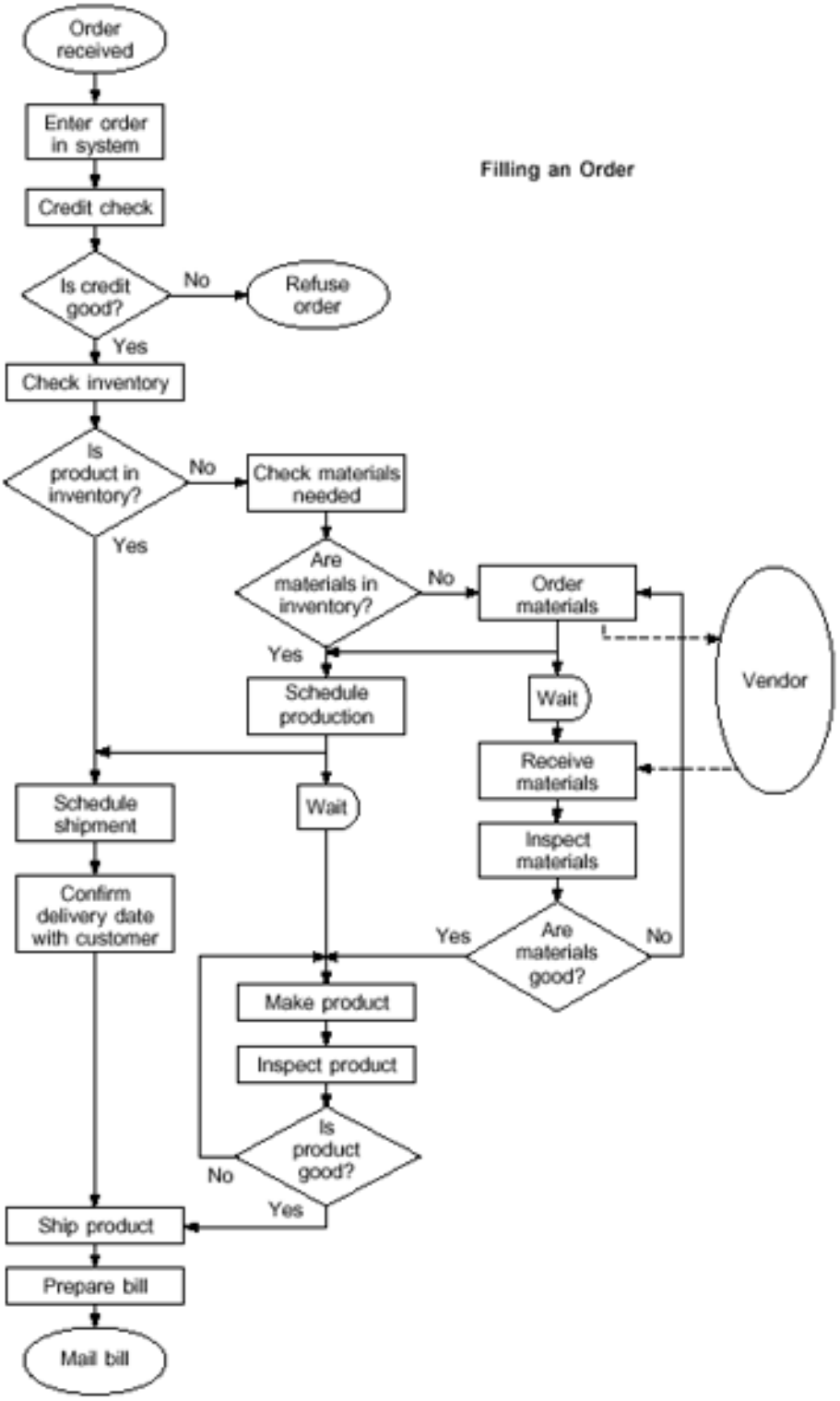
Given below is an example of High-Level Flowchart of an Order Filling Process. It provides the most important steps required in the process.



- **Detailed Flowchart:**

The detailed flowchart provides a detailed picture of a process by mapping all of the steps and activities that occur in the process. This type of flowchart indicates the steps or activities of a process and includes such things as decision points, waiting periods, tasks that frequently must be redone (rework), and feedback loops. This type of flowchart is useful for examining areas of the process in detail and for looking for problems or areas of inefficiency.

Given below is the Detailed Flowchart of an Order Filling Process which shows the sub-steps involved in the process and also reveals the delays that occur when the materials required are not available in the inventory.



Software:

Any drawing program can be used to create flowchart diagrams. Some tools offer special support for flowchart drawing. Now-a-days many software packages exist that can create flowcharts automatically, either directly from source code, or from a flowchart description language. On-line Web-based versions of such programs are available.

Check Your Progress:

1) Write an algorithm for finding N factorial and draw the corresponding flowchart.
   [Where N! = 1*2*3*.....*(N-1)*N]

2) Draw flowchart to find the sum of first 10 natural numbers.

1.5 PROGRAM DESIGN

Program Design is the phase of computer program development in which the hardware and software resources needed by the program are identified and the logic to be used by the program is determined. *Program Design* consists of the steps a programmer should do before they start coding the program in a specific language. These steps when properly documented will make the completed program easier for other programmers to maintain in the future.

The programming process is similar in approach and creativity to writing a paper. In composition, you are writing to express ideas; in programming you are expressing a computation. Both the programmer and the writer must adhere to the syntactic rules (grammar) of a particular language. In prose, the fundamental idea-expressing unit is the sentence; in programming, two units - statements and comments - are available.

An important difference between programming and composition is that in programming you are writing for two audiences: people and computers. As for the computers, what you write is “read” by interpreters and compilers specific to the language you used. They are very rigid about syntactic rules, and perform exactly the calculations you say.

Humans demand even more from programs. This audience consists of two main groups, whose goals can conflict. The larger of the two groups consists of users. Users care about how the program presents itself, its user interface, and how quickly the program runs, how efficient it is. To satisfy this audience, programmers may use statements that are overly terse because they know how to make the program more readable by the computer’s compiler, enabling the compiler to produce faster, but less human-intelligible program. This approach causes the other portion of the audience, i.e. programmers, to boo and hiss. The smaller audience, of which you are also a member, must be able to read the program so that they can enhance and/or change it.

A characteristic of programs is that you and others will seek to modify your program in the future. The program’s meaning is conveyed by statements, and is what the computer interprets. Humans

read this part, which in virtually all languages bears a strong relationship to mathematical equations, and also read comments. Comments are not read by the computer at all, but are there to help explain what might be expressed in a complicated way by programming language syntax. The document or program you write today should be understandable tomorrow, not only by you, but also by others. The program's organization should be easy to follow and the way you write the program, using both statements and comments, should help you and others understand how the computation proceeds. The existence of comments permits the writer to directly express the program's outline in the program to help the reader comprehend the computation.

As a consequence, program design must be extremely structured, having the ultimate intentions of performing a specific calculation efficiently with attractive, understandable, efficient programs. Achieving these general goals means breaking the program into components, writing and testing them separately, then merging them according to the outline.

1.5.1 Activities involved in program design:

There are three broad areas of activities that are considered during program design:

- Understanding the Program
- Using Design Tools to Create a Model
- Develop Test Data

Understanding the Program:

If you are working on a project as a one of many programmers, the system analyst may have created a variety of documentation items that will help you understand what the program is to do. These could include screen layouts, narrative descriptions, documentation showing the processing steps, etc. Understanding the purpose of a program usually involves understanding it's:

- Inputs
- Processing
- Outputs

This IPO approach works very well for beginning programmers. Sometimes, it might help to visualize the programming running on the computer. You can imagine what the monitor will look like, what the user must enter on the keyboard and what processing or manipulations will be done.

Using Design Tools to Create a Model:

At first you will not need a hierarchy chart because your first programs will not be complex. But as they grow and become more

complex, you will divide your program into several modules (or functions).

The first modelling tool you will usually learn is pseudocode. You will document the logic or algorithm of each function in your program. At first, you will have only one function, and thus your pseudocode will follow closely the IPO approach above.

There are several methods or tools for planning the logic of a program. They include: flowcharting, hierarchy or structure charts, pseudocode, etc. Programmers are expected to be able to understand and do flowcharting and pseudocode. Several standards exist for flowcharting and pseudocode and most are very similar to each other. However, most companies have their own documentation standards and styles. Programmers are expected to be able to quickly adapt to any flowcharting or pseudocode standards for the company at which they work.

Understanding the logic and planning the algorithm on paper before you start to code is very important concept. Many students develop poor habits and skipping this step is one of them.

Develop Test Data:

Test data consists of the user providing some input values and predicting the outputs. This can be quite easy for a simple program and the test data can be used to check the model to see if it produces the correct results.

The ***fundamental components*** of good program design are:

1) Problem definition, leading to a program specification,
2) Modular program design, which refines the specification,
3) Module composition, which translates specification into executable program,
4) Module/program evaluation and testing, during which you refine the program and find errors,
5) Program documentation, which pervades all other phases.

The result of following these steps is an efficient, easy-to-use program that has a user's guide (how does someone else run your program) and internal documentation so that other programmers can decipher the algorithm.

In employing programming languages to create software there are distinctly different approaches available. The two most common ones are "procedural programming" and "object-oriented programming". The two approaches differ in the way that the software development and maintenance are planned and implemented. Procedures may use objects, and objects usually use procedures, called methods. Usually the object-oriented code takes more planning and is significantly larger, but it is generally accepted to be easier to

maintain. Today when one can have literally millions of users active for years or decades, maintenance considerations are very important.

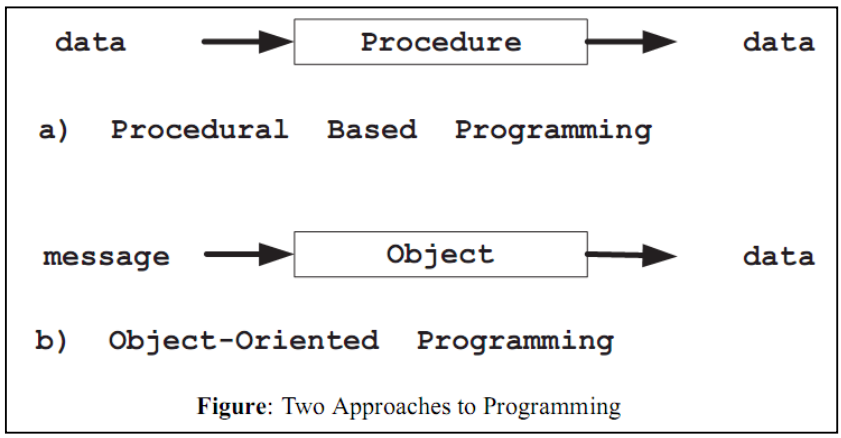


Figure: Two Approaches to Programming

1.5.2 Object-Oriented Formulations:

The process of creating an "object-oriented" (OO) formulation in program design involves at least three stages: Object-Oriented Analysis (OOA), Object-Oriented Design (OOD), and Object-Oriented Programming (OOP).

Steps to be considered for OOA and OOD procedures are given below. Also listed below are seven steps necessary to achieve object-orientedness in an implementation language.

1.5.2.1 Steps in Object-Oriented Analysis:

- Find objects and classes:
  - Create an abstraction of the problem domain.
  - Give attributes behaviours, classes, and objects meaningful names.
  - Identify structures pertinent to the system's complexity and responsibilities.
  - Observe information needed to interact with the system, as well as information to be stored.
  - Look for information re-use; are there multiple structures; can sub-systems be inherited?

- Define the attributes:
  - Select meaningful names.
  - Describe the attribute and any constraints.
  - What knowledge does it possess or communicate?
  - Put it in the type or class that best describes it.
  - Select accessibility as public or private.
  - Identify the default, lower and upper bounds.
  - Identify the different states it may hold.
  - Note items that can either be stored or re-computed.

- Define the behaviour:
  - Give the behaviours meaningful names.
  - What questions should each be able to answer?
  - What services should it provide?
  - Which attribute components should it access?
  - Define its accessibility (public or private).
  - Define its interface prototype.
  - Define any input/output interfaces.
  - Identify a constructor with error checking to supplement the intrinsic constructor.
  - Identify a default constructor.

- Diagram the system:
  - Employ an OO graphical representation.

1.5.2.2 Steps in Object-Oriented Design:

- Improve and add to the OOA results during OOD.
- Divide the member functions into constructors, accessors, agents and servers.
- Design the human interaction components.
- Design the task management components.
- Design the data management components.
- Identify operators to be overloaded.
- Identify operators to be defined.
- Design the interface prototypes for member functions and for operators.
- Design code for re-use through "kind of" and "part of" hierarchies.
- Identify base classes from which other classes are derived.
- Establish the exception handling procedures for all possible errors.

1.5.2.3 Steps to achieve Object-Orientedness:

1 Object-based modular structure:
  - Systems are modularized on the basis of their data structure.

2 Data Abstraction:
  - Objects should be described as implementations of abstract data types.

3 Automatic memory management:
  - Unused objects should be de-allocated by the language system.

4 Classes:
  - Every non-simple type is a module, and every high-level module is a type.

5 Inheritance:

- A class may be defined as an extension or restriction of another.

6 Polymorphism and dynamic binding:
- Entities are permitted to refer to objects of more than one class and operations can have different realizations in different classes.

7 Multiple inheritances:
- Can declare a class as heir to more than one class.

1.6 SUMMARY

- *Algorithm* is the sequence of steps to be performed in order to solve a problem by the computer.
- Three reasons for using algorithms are *efficiency, abstraction* and *reusability.*
- Algorithms can be expressed in many *different notations*, including natural languages, pseudocode, flowcharts and programming languages.
- *Analysis of algorithms* is the theoretical study of computer program performance and resource usage, and is often practised abstractly without the use of specific programming language or implementation.
- The practical goal of *algorithm analysis* is to predict the performance of different algorithms in order to guide program design decisions.
- Most algorithms do not perform the same in all cases; normally an algorithm's performance varies with the data passed to it. Typically, three cases are recognized: the *best case, average case and worst case*.
- *Worst case analysis* of algorithms is considered to be crucial to applications such as games, finance and robotics.
- *O-notation*, also known as *Big O-notation*, is the most common notation used to express an algorithm's performance in a formal manner.
- *Flowchart* is a graphical or symbolic representation of an algorithm. It is the diagrammatic representation of the step-by-step solution to a given problem.
- *Flowcharts* are used in analyzing, designing, documenting or managing a process or program in various fields.
- *Benefits of using flowcharts* include ease of communication, effective and efficient analysis and coding, proper documentation and maintenance.
- *Limitations of using flowcharts* include complex logic and multiple modifications.
- The types of flowcharts are *High-Level Flowchart and Detailed Flowchart.*
- *Program Design* consists of the steps a programmer should do before they start coding the program in a specific language.

- *Program design* must be extremely structured, having the ultimate intentions of performing a specific calculation efficiently with attractive, understandable, efficient programs.
- Three broad areas of activities in program design are *1) Understanding the Program, 2) Using Design Tools to Create a Model and 3) Develop Test Data*
- The process of creating an OO formulation in program design involves at three stages: *1) OOA, 2) OOD, and 3) OOP*.

1.7 UNIT END EXERCISES

1.7.1 Questions:

These questions are intended as a self-test for readers.

1) Define an algorithm and state the benefits and reasons of using algorithms?

2) Explain the common approaches used in designing an algorithm?

3) What is algorithm analysis and why is it important?

4) Explain worst-case analysis of algorithms?

5) Explain O-notation with an example?

6) State the rules for O-notation?

7) What is a flowchart? Explain with suitable examples.

8) State the advantages and limitations of using flowcharts?

9) What is program design? Explain the steps involved in program design.

10) State the fundamental components of a good program design?

11) What are the steps involved in Object-Oriented Analysis?

12) What are the steps involved in Object-Oriented Design?

1.7.2 Programming Projects:

Writing programs that solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied.

1) Write an algorithm to find the sum of first 50 natural numbers and also draw the corresponding flowchart.

2) Write an algorithm to read a number N from the user and print all its divisors.

3) Write an algorithm to find the sum of given N numbers and also draw the corresponding flowchart.

4) Write an algorithm to compute the sum of the squares of integers from 1 to 50 and also draw the corresponding flowchart. Assuming that you have to code the above given problem, show the steps involved in program design.

5) Draw a flowchart explaining the process of waking up in the morning. [Hint: Use steps Alarm Ringing, Ready to get up, Hitting snooze button, climbing out of bed]

1.8 FURTHER READING

An Introduction to the Analysis of Algorithms by Sedgewick, Robert, and Philippe Flajolet.

Fundamental Algorithms, Third Edition by Knuth, Donald.

Wikipedia page on Algorithm, Flowchart and Program Design.

wikipedia.org/wiki/algorithm

wikipedia.org/wiki/flowchart

wikipedia.org/wiki/program_design

Introduction to Algorithms, Second Edition by Thomas H. Cormen

2

Introduction to C++

Unit Structure:

2.1 OBJECTIVES

After completing this chapter, you will be able to:

- Understand the origin and history of C++
- Identify the applications that use C++
- Understand simple C++ programs
- Identify the pitfalls for C++, and
- Write better C++ programs using programming tips

2.2 INTRODUCTION

The most important thing to do when learning C++ is to focus on concepts and not get lost in language-technical details. The purpose of learning a programming language is to become a better programmer; that is, to become more effective at designing and implementing new systems and at maintaining old ones. For this, an appreciation of programming and design techniques is far more important than an understanding of details; that understanding comes with time and practice.

2.3 ORIGIN AND HISTORY OF C++

C++ is an object oriented programming language. It was developed by Bjarne Stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey, USA, in the early 1980's. Stroustrup, an admirer of Simula67 and a strong supporter of C, wanted to combine the best of both the languages and create a more powerful language that could support object-oriented programming features and still retain the power and elegance of C. The result was C++.

A programming language serves two related purposes: it provides a vehicle for the programmer to specify actions to be executed, and it provides a set of concepts for the programmer to use when thinking about what can be done. The first purpose ideally requires a language that is "close to the machine" so that all important aspects of a machine are handled simply and efficiently in a way that is reasonably obvious to the programmer. The second purpose ideally requires a language that is "close to the problem to be solved" so that the concepts of a solution can be expressed directly and concisely. The facilities added to C to create C++ were primarily designed keeping these purposes in mind.

C++ is regarded as a middle-level language, as it comprises a combination of both high-level and low-level language features. C++ was developed as an enhancement to the C language and originally named C with classes.

The main source of inspiration for C++ was C and hence it was initially called C with classes. C is retained as a subset and also C's emphasis on facilities that are low-level enough to cope with the most demanding systems programming tasks. The other main source of inspiration was Simula67; the class concept (with derived classes and virtual functions) was borrowed from it. C++'s facility for overloading operators and the freedom to place a declaration wherever a statement can occur resembles Algol68. Templates were partly inspired by Ada's generics (both their strengths and weaknesses) and partly by Clu's parameterized modules. Similarly, the C++ exception handling mechanism was inspired partly by Ada, Clu, and ML.

The name C++ (pronounced ''see plus plus'') was coined by Rick Mascitti in 1983. The name signifies the evolutionary nature of the changes from C; ''++'' is the C increment operator. The slightly shorter name ''C+'' is a syntax error; it has also been used as the name of an unrelated language. Connoisseurs of C semantics find C++ inferior to ++C. The language is also not called D, because it is an extension of C, and it does not attempt to remedy problems by removing features.

After years of development, the C++ programming language standard was ratified in 1998 as ISO/IEC 14882:1998. That standard is still current, but is amended by the 2003 technical corrigendum, ISO/IEC 14882:2003.

2.4 APPLICATIONS OF C++

As one of the most popular programming languages ever created, C++ is used by hundreds of thousands of programmers in essentially every application domain. Some of its application domains include systems software, application software, device drivers, embedded software, high-performance server and client applications, and entertainment software such as video games. C++ has greatly influenced many other popular programming languages, most notable C# and Java. C++ is also used for hardware design, where design is initially described in C++. Below a few of its applications are mentioned elaborately:

- Early applications tended to have a strong systems programming flavor. For example, several *major operating systems* have been written in C++ and many more have key parts done in C++.
- C++ provides uncompromising low-level efficiency essential for device drivers and other software that rely on direct manipulation of hardware under real-time constraints. In such code, predictability of performance is at least as important as raw speed.
- Most applications have sections of code that are critical for acceptable performance. For most code, maintainability, ease of extension, and ease of testing is key. C++'s support for these concerns has led to its widespread use where reliability is a must and in areas where requirements change significantly over time. Examples are banking, trading, insurance, telecommunications, and military applications.
- Graphics and user interfaces are areas in which C++ is heavily used. Anyone who has used either an Apple Macintosh or a PC running Windows has indirectly used C++ because the primary user interfaces of these systems are C++ programs. In addition, some of the most popular libraries supporting X for UNIX are written in C++. Thus, C++ is a common choice for the vast number of applications in which the user interface is a major part.

2.5 C AND C++

Although C was restricted to the UNIX programming environment, C++ was never restricted to UNIX. It simply used UNIX and C as a model for the relationships between language, libraries, compilers, linkers, execution environments etc. That minimal model helped C++ to be successful on essentially every computing platform.

Many books will highlight the above mentioned and other differences between C and C++; however the reason for C to be chosen as the base language does go unnoticed by people.

C was chosen as the base language for C++ because it

- is versatile, terse, and relatively low-level;
- is adequate for most systems programming tasks;
- runs everywhere and on everything; and
- fits into the UNIX programming environment.

C has its problems, but a language designed from scratch would have some too, and we know C's problems. Importantly, working with C enabled ''C with Classes'' to be a useful tool within months of the first thought of adding Simula-like classes to C.

Knowing C is not a prerequisite for learning C++. Programming in C encourages many techniques and tricks that are rendered unnecessary by C++ language features. For example, explicit type conversion (casting) is less frequently needed in C++ than it is in C. However, good C programs tend to be C++ programs. For example, every program in ***Kernighan and Ritchie, The C Programming Language (2nd Edition)***, is a C++ program. Experience with any statically typed language will be a help when learning C++.

2.6 A SIMPLE C++ PROGRAM

Before looking at how to write C++ programs consider the following simple example program

```
// my first program in C++

#include <iostream.h>
using namespace std;

int main ()
{
  cout << "Hello World!";
  return 0;
}
```

The previous program is the typical program that programmer apprentices write for the first time, and its result is the printing on screen of the "Hello World!" sentence. It is one of the simplest programs that can be written in C++, but it already contains the fundamental components that every C++ program has. We are going to look line by line at the code we have just written:

```
// my first program in C++
```

This is a comment line. All lines beginning with two slash signs (*//*) are considered comments and do not have any effect on the behaviour of the program. The programmer can use them to include short explanations or observations within the source code itself. In this case, the line is a brief description of what our program is.

```
#include <iostream>
```

Lines beginning with a hash sign (*#*) are directives for the preprocessor. They are not regular code lines with expressions but

indications for the compiler's preprocessor. In this case the directive *#include<iostream>* tells the preprocessor to include the *iostream* standard file. This specific file (*iostream*) includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is going to be used later in the program.

```
using namespace std;
```

All the elements of the standard C++ library are declared within what is called a namespace, the namespace with the name *std*. So in order to access its functionality we declare with this expression that we will be using these entities. This line is very frequent in C++ programs that use the standard library.

```
int main ()
```

This line corresponds to the beginning of the definition of the *main* function. The *main* function is the point by where all C++ programs start their execution, independently of its location within the source code. It does not matter whether there are other functions with other names defined before or after it - the instructions contained within this function's definition will always be the first ones to be executed in any C++ program. For that same reason, it is essential that all C++ programs have a main function. The word main is followed in the code by a pair of parentheses (()). That is because it is a function declaration. In C++, what differentiate a function declaration from other expressions are the parentheses that follow its name. Optionally, these parentheses may enclose a list of parameters within them. Right after these parentheses we can find the body of the main function enclosed in braces ({}). What is contained within these braces is what the function does when it is executed.

```
cout << "Hello World!";
```

This line is a C++ statement. A statement is a simple or compound expression that can actually produce some effect. In fact, this statement performs the only action that generates a visible effect in our first program. *cout* represents the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case the Hello World sequence of characters) into the standard output stream (which usually is the screen). *cout* is declared in the *iostream* standard file within the std namespace, so that's why we needed to include that specific file and to declare that we were going to use this specific namespace earlier in our code. Notice that the statement ends with a semicolon character (;). This character is used to mark the end of the statement and in fact it must be included at the end of all expression statements in all C++ programs (one of the most common syntax errors is indeed to forget to include some semicolon after a statement).

```
return 0;
```

The *return* statement causes the main function to finish. *return* may be followed by a return code (in our example is followed by the

return code 0). A return code of 0 for the main function is generally interpreted as the program worked as expected without any errors during its execution. This is the most usual way to end a C++ console program.

You may have noticed that not all the lines of this program perform actions when the code is executed. There were lines containing only comments (those beginning by //). There were lines with directives for the compiler's pre-processor (those beginning by #). Then there were lines that began the declaration of a function (in this case, the main function) and, finally lines with statements (like the insertion into *cout*), which were all included within the block delimited by the braces ({}) of the main function.

The program has been structured in different lines in order to be more readable, but in C++, we do not have strict rules on how to separate instructions in different lines. For example, instead of

```
int main ()
{
  cout << " Hello World!";
  return 0;
}
```

We could have written:

```
int main () { cout << "Hello World!"; return 0; }
```

All in just one line and this would have had exactly the same meaning as the previous code. In C++, the separation between statements is specified with an ending semicolon (;) at the end of each one, so the separation in different code lines does not matter at all for this purpose. We can write many statements per line or write a single statement that takes many code lines. The division of code in different lines serves only to make it more legible and schematic for the humans that may read it.

At this stage let us consider the general format of a C++ program:

```
// Introductory comment
// file name, programmer, when written or modified
// what program does

#include <iostream.h>

void main()
{
constant declarations
variable declarations
executable statements
}
```

Note that it makes complex programs much easier to interpret if, as above, closing braces (}) are aligned with the corresponding

opening brace ({). However other conventions are used for the layout of braces in textbooks and other C++ programmers' programs. Also additional spaces, new lines etc. can also be used to make programs more readable. The important thing is to adopt one of the standard conventions and stick to it consistently.

The format given above is the general format of a C++ program. Although this structure will also include the class declaration and member functions definitions but we will come elaborate on that part later.

Before looking at how to write C++ programs consider another sample program.

```
// Sample program
// Reads values for the length and width of a rectangle
// and returns the perimeter and area of the rectangle.

#include <iostream.h>

void main()
{
  int length, width;
  int perimeter, area;              // declarations
  cout <<  "Length = ";             // prompt user
  cin >> length;                    // enter length
  cout << "Width = ";               // prompt user
  cin >> width;                     // input width
  perimeter  =  2*(length  +  width);          //  compute
perimeter
  area = length * width;              // compute area
  cout << endl
       << "Perimeter is " << perimeter;
  cout << endl
       << "Area is " << area
       << endl;                    // output results
} // end of main program
```

The following points should be noted in the above program:

1) Any text from the symbols // until the end of the line is ignored by the compiler. This facility allows the programmer to insert **Comments** in the program. Every program should at least have a comment indicating the programmer's name, when it was written and what the program actually does. Any program that is not very simple should also have further comments indicating the major steps carried out and explaining any particularly complex piece of programming. This is essential if the program has to be amended or corrected at a later date.

2) The line

   ```
   #include <iostream.h>
   ```

   causes the compiler to include the text of the named file (in this case *iostream.h*) in the program at this point. The file *iostream.h* is a system supplied file which has definitions in it which are required if the program is going to use stream input

or output. All your programs will include this file. This statement is a **compiler directive** -- that is it gives information to the compiler but does not cause any executable code to be produced.

3) The actual program consists of the **function** main which commences at the line

```
void main()
```

All programs must have a function main. Note that the opening brace ({) marks the beginning of the body of the function, while the closing brace (}) indicates the end of the body of the function. The word void indicates that main does not return a value. Running the program consists of obeying the statements in the body of the function main.

4) The body of the function main contains the actual code which is executed by the computer and is enclosed, as noted above, in braces {}.

5) Every statement which instructs the computer to do something is terminated by a semi-colon. Symbols such as main(), { } etc. are not instructions to do something and hence are not followed by a semi-colon.

6) Sequences of characters enclosed in double quotes are literal strings. Thus instructions such as

```
cout << "Length = "
```

send the quoted characters to the output stream *cout*. The special identifier *endl* when sent to an output stream will cause a newline to be taken on output.

7) All variables that are used in a program must be declared and given a type. In this case all the variables are of type int, i.e. whole numbers. Thus the statement

```
int length, width;
```

declares to the compiler that integer variables length and width are going to be used by the program. The compiler reserves space in memory for these variables.

8) Values can be given to variables by the **assignment** statement, e.g. the statement

```
area = length*width;
```

evaluates the expression on the right-hand side of the equals sign using the current values of length and width and assigns the resulting value to the variable area.

9) Layout of the program is quite arbitrary, i.e. new lines, spaces etc. can be inserted wherever desired and will be ignored by the compiler. The prime aim of additional spaces, new lines, etc. is

to make the program more readable. However superfluous spaces or new lines must not be inserted in words like *main*, *cout*, in variable names or in strings (unless you actually want them printed).

2.7 PROGRAMMING TIPS

A. Put the constant on the left in a conditional:

We've all experienced bugs like this:

```
while (continue = TRUE)
{
// ...this loops forever!
}
```

This type of problem can be solved by putting the constant on the left, so if you leave out an = in a conditional, you will get a compiler error instead of a program bug (because constants are non lvalues, of course):

```
while (TRUE = continue)
{
// compile error!
}
```

B Use exceptions:

Use the try/throw/catch mechanisms in C++ - they are very powerful. Many people implement an exception class, which they use for general error reporting throughout their program.

```
class ProgramException
{
// pass in a pointer to string, make sure string still
exists when
// the PrintError() method is called
      ProgramException(const  char*  const  szErrorMsg  =
NULL)
      {
            if (NULL == szErrorMsg)
                  m_szMsg = "Unspecified error";
            else
                  m_szMsg = szErrorMsg;
      };

      void PrintError()
      {
            cerr << m_szMsg << endl;
      };
};

void OpenDataFile(const char* const szFileName)
{
      assert(szFileName);
      if (NULL == fopen(szFileName, "r"))
      throw ProgramException("File not found");
```

```
        // ...
}

int main(void)
{
        try
        {
                OpenDataFile("foo.dat");
        }
        catch (ProgramException e)
        {
                e.PrintError();
                return EXIT_FAILURE;
        }
        return EXIT_SUCCESS;
}
```

C. Virtual functions:

In C++ virtual functions are used for enabling Polymorphism. Following two important points need to be noted while using virtual functions:

- A function declared as virtual in the base class should to be declared as virtual in the derived class as well.
- A class which has a member function declared as virtual needs to have its destructor to be defined as virtual as well. This is required for proper calls to the destructor up the class hierarchy.

D. Don't ignore API function return values:

Most API functions will return a particular value which represents an error. You should test for these values every time you call the API function. If you don't want to clutter your code with error-testing then wrap the API call in another function (do this when you are thinking about portability too) which tests the return value and either asserts, handles the problem, or throws an exception. The above example of *OpenDataFile* is a primitive way of wrapping *fopen* with error-checking code which throws an exception if *fopen* fails.

E. Be consistent:

Be consistent in the way you write your code. Use the same indentation and bracketing style everywhere. If you put the constant on the left in a conditional, do it everywhere. If you assert on your pointers, do it everywhere. Use the same kind of comment style for the same kind of comments. If you are the type to go in for a naming convention (like Hungarian notation), then you have to stick to it everywhere. Don't do *int iCount* in one place and *int nCount* in another.

F. Identifier clashes between source files:

In C++, variables and functions are by default public, so that any C++ source file may refer to global variables and functions from another C source file. This is true even if the file in question does not have a declaration or prototype for the variable or function. You must, therefore, ensure that the same symbol name is not used in two different files. If you don't do this you will get linker errors and possibly warnings during compilation.

2.8 SUGGESTIONS FOR C PROGRAMMERS

The better one knows C, the harder it seems to be to avoid writing C++ in C style, thereby losing some of the potential benefits of C++. Here are a few pointers to the areas in which C++ has better ways of doing something than C has:

1) ***Macros*** are almost never necessary in C++. Use ***const*** or ***enum*** to define manifest constants, ***inline*** to avoid function-calling overhead, ***templates*** to specify families of functions and types, and ***namespaces*** to avoid name clashes.

2) Don't declare a variable before you need it so that you can initialize it immediately. A declaration can occur anywhere a statement can, in for-statement initializers, and in conditions.

3) Don't use ***malloc*** (). The ***new*** operator does the same job better, and instead of ***realloc*** (), try a ***vector***.

4) Try to avoid ***void****, pointer arithmetic, unions, and casts, except deep within the implementation of some function or class. In most cases, a cast is an indication of a design error. If you must use an explicit type conversion, try using one of the ''new casts'' for a more precise statement of what you are trying to do.

5) Minimize the use of arrays and C-style strings. The C++ standard library ***string*** and ***vector*** classes can often be used to simplify programming compared to traditional C style. In general, try not to build yourself what has already been provided by the standard library.

Most important, try thinking of a program as a set of interacting concepts represented as classes and objects, instead of as a bunch of data structures with functions twiddling their bits.

2.9 PITFALLS

Pitfall is a C++ code that compiles, links, runs but does something different than you expect. This is an attempt to provide an overview of many of the C++ pitfalls that beg content to moderately experienced C++ programmers often fail to understand. Please note that this is not an attempt to replace a good C++ reference, but simply

to provide an introduction to some often misunderstood concepts and to point out their usefulness.

Example:

```
if (-0.5 <= x <= 0.5) return 0;
```

Pitfall:

```
if (-0.5 <= x <= 0.5) return 0;
```

This expression does *not* test the mathematical condition

```
-0.5 <= x <= 0.5
```

Instead, it first computes `-0.5 <= x`, which is 0 or 1, and then compares the result with 0.5.

Even though C++ now has a `bool` type, Booleans are still freely convertible to `int`. Since `bool->int` is allowed as a conversion, the compiler cannot check the validity of expressions.

A. References:

References are a way of assigning a "handle" to a variable. There are two places where this is used in C++. We'll discuss both of them briefly.

Assigning References:

This is the less often used variety of references, but still worth noting as an introduction to the use of references in function arguments. Here we create a reference that looks and acts like a standard C++ variable except that it operates on the same data as the variable that it references.

```
int foo = 3;      // foo == 3
int &bar = foo;   // foo == 3
bar = 5;          // foo == 5
```

Here because we've made *bar* a reference to *foo* changing the value of bar also changes the value of *foo*.

Passing Function Arguments with References:

The same concept of references is used when passing variables. For example:

```
void foo( int &i )
{
    i++;
}
```

```
int main()
{
    int bar = 5;    // bar == 5
    foo( bar );     // bar == 6
    foo( bar );     // bar == 7

    return 0;
}
```

Here we display one of the two common uses of references in function arguments — they allow us to use the conventional syntax of passing an argument by value but manipulate the value in the caller.

Note: While sometimes useful, using this style of references can sometimes lead to counter-intuitive code. It is not clear to the caller of *foo()* above that *bar* will be modified without consulting an API reference.

However there is a more common use of references in function arguments — they can also be used to pass a handle to a large data structure without making multiple copies of it in the process. Consider the following:

```
void foo( const std::string &s )
{
    std::cout << s << std::endl;
}

void bar( std::string s )
{
    std::cout << s << std::endl;
}

int main()
{
    std::string text = "This is a test.";

    foo( text ); // doesn't make a copy of "text"
    bar( text ); // makes a copy of "text"

    return 0;
}
```

In this simple example we're able to see the differences in *pass by value* and *pass by reference*. In this case pass by value just expends a few additional bytes, but imagine instance if *text* contained the text of an entire book. The ability to pass it by reference keeps us from needing to make a copy of the string and avoids the ugliness of using a pointer.

It should also be noted that this only makes sense for *complex* types — *classes* and *structs*. In the case of *ordinal types* — i.e. *int, float, bool,* etc. — there is no savings in using a reference instead of simply using *pass by value*.

B. Public, Protected and Private Labels:

C++ supports three labels that can be used in classes (or structs) to define the permissions for the members in that section of the class. These labels can be used multiple times in a class declaration for cases where it's logical to have multiple groups of these types. These keywords affect the permissions of the members — whether functions or variables.

Public: *This label is used to say that the methods and variables may be accessed from any portion of the code that knows the class type. This should usually only be used for member functions (and not variables) and should not expose implementation details.*

Protected: *Only subclasses of this type may access these functions or variables. Many people prefer to also keep this restricted to functions (as opposed to variables) and to use accessor methods for getting to the underlying data.*

Private: *This is used for methods that cannot be used in either subclasses or other places. This is usually the domain of member variable and helper functions. It's often useful to start off by putting functions here and to only move them to the higher levels of access as they are needed.*

It's often misunderstood that different instances of the same class may access each others' private or protected variables. A common case for this is in copy constructors.

```
class Foo
{
public:
    Foo( const Foo &f )
    {
        m_value = f.m_value; // perfectly legal
    }

private:
    int m_value;
}
```

(It should however be mentioned that the above is not needed as the default copy constructor will do the same thing.)

C. Constructor pitfalls:

D. *Example:*

```
E.              int main()
                {
F.          string a("Hello");
G.          string b();
H.          string c = string("World");
I.          // ...
                return 0;
            }
```

J. *Pitfall:*

```
K.              string b();
```

This expression does not construct an object `b` of type `string`. Instead, it is the prototype for a function `b` with no arguments and return type `string`. Remember to omit the `( )` when invoking the default constructor. The C feature of declaring a function in a local scope is worthless since it lies about the true scope. Most programmers place all prototypes in header files. But even a worthless feature that you never use can haunt you.

2.10 TESTING AND DEBUGGING

There are lots of error conditions that happen in the normal life of a program. For instance, file not found, out of memory, or invalid user input. You should always handle these conditions gracefully (by re-prompting for a filename, by freeing memory or telling the user to quit other applications, or by telling the user there is an error in his input, respectively). However, there are other conditions which are not real error conditions, but are the result of bugs. For example, say you have a routine which copies a string into a buffer, and no one is supposed to pass in a NULL pointer to the routine. You do not want to do something like this.

Let us look at a few of the errors that occur in C++ programs:

- ***Syntax errors*** – These errors are found by the compiler when you compile your C++ program. Syntax errors often cascade; correcting the first error may eliminate several following errors. Many programmers try to correct these errors through guess work which seldom works, however it does waste time. Most of the time the actual error often occurs in the preceding line and not in the line shown by the compiler.
- ***Logical errors*** – These errors are hard to find. In order to find these errors the entire logic of the program needs to be checked and revisited. Sometimes the algorithms and the flowcharts drawn for the problem also need to be checked.

Correcting the above errors involves testing and debugging the code. Testing the code involves testing each function separately using stubs and drivers. Debugging involves displaying intermediate results which helps in finding the bug in the program, if any.

Drivers allow you to test a function without the rest of the program. Drivers help execute a function and show the result. Using a loop allows you to retest the function on different arguments without re-running the program.

```
//Driver program for testing swap_values
#include <iostream.h>
void swap_values(int& var1, int& var2);
int main()
{
```

```
        int first, second;
        char ans = 'y';

        do
        {
        cout << "Enter two integers: ";
        cin >> first >> second;
        swap_values(first, second);
        cout << "Reverse order of the numbers: "
             << first << " " << second << endl;
        cout << "Test again? (Type y for yes)";
        cin >> ans;
        cout << endl;
        } while (ans == 'Y' || ans == 'y');
        return 0;
}

void swap_values(int& var1, int& var2)
{
        int temp;
        temp = var1;
        var1 = var2;
        var2 = temp;
}
```

Stubs are utterly simple substitutes for a function, so you can test the call itself without having to worry about getting the function correct (yet). Stubs are replaced by functions one at a time. Stubs are simple enough and give you confidence for bug-free code.

```
//Stub program for testing swap_values
int main()
{
        int a, b;
        ......
        ......
        age_of_universe(a,b);
        ......
        ......
        return 0;
}

void age_of_universe(int& x, int& y)
{
        x = x + 1;
        y = y + 2;
        //  it's really much more complex than this
        cout << "Inside age_of_universe" << endl;
}
```

2.11 Summary

- C++ is an object oriented programming language developed by Bjarne Stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey, USA, in the early 1980's.

- C++ is regarded as a middle-level language, as it comprises a combination of both high-level and low-level language features.
- C++ was developed as an enhancement to the C language and originally named C with classes.
- Some of the application domains of C++ include systems software, application software, device drivers, embedded software, high-performance server and client applications, and entertainment software such as video games.
- C++ has greatly influenced many other popular programming languages, most notable C# and Java. C++ is also used for hardware design, where design is initially described in C++
- Comments are ignored by the compiler but are there for the information of someone reading the program. All characters between `//` and the end of the line are ignored by the compiler.
- All programs must include a function *main ()*. The body of the function main contains the actual code which is executed by the computer and is enclosed in braces {}.
- Every statement (executable statement) which instructs the computer to do something is terminated by a semi-colon.
- Be consistent in the way you write your code. Use the same indentation and bracketing style everywhere.
- Pitfall is a C++ code that compiles, links, runs but does something different than you expect.
- C++ supports three labels (public, private and protected) that can be used in classes (or structs) to define the permissions for the members in that section of the class.
- Testing the code involves testing each function in order to get the desired output.
- Debugging involves how to find a bug by displaying intermediate results.

2.12 UNIT END EXERCISES

These questions are intended as a self-test for readers.

2.12.1 Fill in the blanks:

1) C++ was developed by _____ at _____.

2) C++ is regarded as _____ language.

3) C++ was earlier called _____.

2.12.2 Questions:

1) State the application domains that use C++?

2) What is a pitfall? Explain any one pitfall elaborately.

3) What is the purpose of the following two lines:

```
#include <iostream>
using namespace std;
```

4) Explain Syntax and Logical errors?

2.13 FURTHER READING

Scott Meyers. Effective C++: 55 Specific Ways to Improve Your Programs and Designs. Addison-Wesley, third edition, 2007

The C++ FAQ: http://www.parashift.com/c++-faq-lite/

http://www.gmonline.demon.co.uk/cscene/CS2/CS2-01.html

http://autotoolset.sourceforge.net/tutorial.html

Bjarne Stroustrup, "The C++ Programming Language: Second Edition".

3

Chapter 3
Variables and Assignments

Unit Structure:

3.1 OBJECTIVES

After completing this chapter you will be able to

- Distinguish between variable and constant
- Identify the keywords in C++
- Declare and Initialize variables
- Identify scope of variables in a given program
- Use reference variables.

3.2 INTRODUCTION

The usefulness of the "Hello World" programs shown in the previous section is quite questionable. We had to write several lines of

code, compile them, and then execute the resulting program just to obtain a simple sentence written on the screen as result. It certainly would have been much faster to type the output sentence by ourselves. However, programming is not limited only to printing simple texts on the screen. In order to go a little further on and to become able to write programs that perform useful tasks that really save us work we need to introduce the concept of variable.

3.3 VARIABLES

A variable is a symbolic name for a memory location in which data can be stored and subsequently recalled. Variables are used for holding data values so that they can be utilized in various computations in a program. All variables have two important attributes:

- A type which is established when the variable is defined (e.g., integer, real, character). Once defined, the type of a C++ variable cannot be changed.
- A value which can be changed by assigning a new value to the variable. The kind of values a variable can assume depends on its type. For example, an integer variable can only take integer values (e.g., 2, 100, -12).

A variable is used for the quantities which are manipulated by a computer program. For example a program that reads a series of numbers and sums them will have to have a variable to represent each number as it is entered and a variable to represent the sum of the numbers.

Let us illustrate the uses of some simple variable.

```
#include <iostream.h>
int main (void)
{
      int workDays;
      float workHours, payRate, weeklyPay;
      workDays = 5;
      workHours = 7.5;
      payRate = 38.55;
      weeklyPay = workDays  * workHours * payRate;
      cout << "Weekly Pay = ";
      cout << weeklyPay;
      cout << '\n';
}
```

The above given program calculates the weekly pay using three variables workDays, workHours, and payRate.

In order to distinguish between different variables, they must be given **identifiers.** Each variable needs an identifier that distinguishes it from the others, for example, in the previous code the variable identifiers were weeklyPay, workDays, workHours and payRate, but we could have called the variables any names we wanted to invent, as long as they were valid identifiers.

3.4 IDENTIFIERS

Identifiers are names given to variables which distinguish them from all other variables. The rules of C++ for valid identifiers state that:

An identifier must:

- start with a letter
- consist only of letters, the digits 0-9, or the underscore symbol _
- Not be a **reserved word**.

Reserved words are otherwise valid identifiers that have special significance to C++. For the purposes of C++ identifiers, the underscore symbol, _, is considered to be a letter. Its use as the first character in an identifier is not recommended though, because many library functions in C++ use such identifiers. Similarly, the use of two consecutive underscore symbols, __, is forbidden.

The following are valid identifiers:

`Length, days_in_year, DataSet1, Profit95, _Pressure, first_one, first_1` although using `_Pressure` is not recommended.

The following are invalid:

```
days-in-year 1data int first.val throw
```

Identifiers should be chosen to reflect the significance of the variable in the program being written. Although it may be easier to type a program consisting of single character identifiers, modifying or correcting such a program becomes more and more difficult. The minor typing effort of using *meaningful* identifiers will repay itself many fold in the avoidance of simple programming errors when the program is modified.

At this stage it is worth noting that C++ is **case-sensitive**. That is lower-case letters are treated as distinct from upper-case letters. Thus the word `days` in a program is quite different from the word `Days` or the word `DAYS`.

3.5 RESERVED WORDS

The **syntax rules** (or grammar) of C++ define certain symbols to have a unique meaning within a C++ program. These symbols, the **reserved words**, must not be used for any other purposes. Reserved words are otherwise valid identifiers that have special significance to C++. All reserved words are in lower-case letters. The table below lists the reserved words of C++.

C++ Keywords					
and	*and_eq*	*asm*	*auto*	*bitand*	*bitor*
bool	*break*	*case*	*catch*	*char*	*class*
compl	*const*	*const_cast*	*continue*	*default*	*delete*
do	*double*	*dynamic_cast*	*else*	*enum*	*explicit*
export	*extern*	*false*	*float*	*for*	*friend*
goto	*if*	*inline*	*int*	*long*	*mutable*
namespace	*new*	*not*	*not_eq*	*operator*	*or*
or_eq	*private*	*protected*	*public*	*register*	*reinterpret_cast*
return	*short*	*signed*	*sizeof*	*static*	*static_cast*
struct	*switch*	*template*	*this*	*throw*	*true*
try	*typedef*	*typeid*	*typename*	*union*	*unsigned*
using	*virtual*	*void*	*volatile*	*wchar_t*	*while*
xor	*xor_eq*				

Some of these reserved words may not be treated as reserved by older compilers. However you would do well to avoid their use. Other compilers may add their own reserved words. Typical are those used by Borland compilers for the computer, which add `near`, `far`, `huge`, `cdecl`, and `pascal`.

Notice that `main` is *not* a reserved word. However, this is a fairly technical distinction, and for practical purposes you are advised to treat `main`, `cin`, and `cout` as if they were reserved as well.

3.6 DECLARATION OF VARIABLES

In C++ (as in many other programming languages) all the variables that a program is going to use must be **declared** prior to use. Declaration of a variable serves two purposes:

- It associates a **type** and an identifier (or name) with the variable. The type allows the compiler to interpret statements correctly. For example in the CPU the instruction to add two integer values together is different from the instruction to add two floating-point values together. Hence the compiler must know the type of the variables so it can generate the correct add instruction.
- It allows the compiler to decide how much storage space to allocate for storage of the value associated with the identifier and to assign an address for each variable which can be used in code generation.

A typical set of variable declarations that might appear at the beginning of a program could be as follows:

```
int i, j, count;
float sum, product;
char ch;
```

```
bool passed_exam;
```

which declares integer variables `i`, `j` and `count`, real variables `sum` and `product`, a character variable `ch`, and a boolean variable `pass_exam`.

A variable declaration has the form:

type identifier-list;

type specifies the type of the variables being declared. The *identifier-list* is a list of the identifiers of the variables being declared, separated by commas.

Variables may be initialised at the time of declaration by assigning a value to them as in the following example:

```
int i, j, count = 0;
float sum = 0.0, product;
char ch = '7';
bool passed_exam = false;
```

which assigns the value 0 to the integer variable `count` and the value 0.0 to the real variable `sum`. The character variable `ch` is initialised with the character `7`. `i`, `j`, and `product` have no initial value specified, so the program should make *no* assumption about their contents.

To see what variable declarations look like in action within a program, we are going to see the following C++ code.

```
// operating with variables

#include <iostream>
using namespace std;

int main ()
{
        // declaring variables:
        int a, b;
        int result;

        // process:
        a = 5;
        b = 2;
        a = a + 1;
        result = a - b;

        // print out the result:
        cout << result;

        // terminate the program:
  return 0;
}
```

Do not worry if something else than the variable declarations themselves looks a bit strange to you. You will see the rest in detail in coming sections.

3.7 SCOPE OF VARIABLES

A declaration introduces a name into a *scope*; that is, a name can be used only in a specific part of the program text. For a name declared in a function (often called a *local* name), that scope extends from its point of declaration to the end of the block in which its declaration occurs. A block is a section of code delimited by a { } pair.

A name is called *global* if it is defined outside any function, class, or namespace. The scope of a global name extends from the point of declaration to the end of the file in which its declaration occurs. A declaration of a name in a block can hide a declaration in an enclosing block or a global name. That is, a name can be redefined to refer to a different entity within a block. After exit from the block, the name resumes its previous meaning. For example:

```
int x;                  // global x
void f()
{
        int x;          // local x hides global x
        x = 1;          // assign to local x
        {
                int x;  // hides first local x
                x = 2;  // assign to second local x
        }
        x = 3;          // assign to first local x
}
int *p = &x;            // take address of global x
```

Hiding names is unavoidable when writing large programs. However, a human reader can easily fail to notice that a name has been hidden. Because such errors are relatively rare, they can be very difficult to find. Consequently, name hiding should be minimized. Using names such as i and x for global variables or for local variables in a large function is asking for trouble.

A hidden global name can be referred to using the scope resolution operator (: :). For example:

```
int x;
void f2()
{
        int x = 1;              // hide global x
        :  :x = 2;             // assign to global x
        x = 2;          // assign to local x
        // ...
}
```

There is no way to use a hidden local name.
The scope of a name starts at its point of declaration; that is, after the complete declarator and before the initializer.

3.8 INITIALIZATION OF VARIABLES

When declaring a regular local variable, its value is by default undetermined. But you may want a variable to store a concrete value at the same moment that it is declared. In order to do that, you can initialize the variable. There are two ways to do this in C++:

The first one, known as c-like, is done by appending an equal sign followed by the value to which the variable will be initialized:

```
type identifier = initial_value ;
```

For example, if we want to declare an int variable called a initialized with a value of 0 at the moment in which it is declared, we could write:

```
int a = 0;
```

The other way to initialize variables, known as constructor initialization, is done by enclosing the initial value between parentheses
(()):

```
type identifier (initial_value) ;
```

For example:

```
int a (0);
```

Both ways of initializing variables are valid and equivalent in C++.

```
// initialization of variables

#include <iostream>
using namespace std;

int main ()
{
        int a=5;               // initial value = 5
        int b(2);              // initial value = 2
        int  result;                    //  initial  value
undetermined

        a = a + 3;
        result = a - b;
        cout << result;

        return 0;
}
```

3.9 REFERENCE VARIABLES

C++ allows you to create a second name for the variable that you can use to read or modify the original data stored in that variable. While this may not sound appealing at first, what this means is that when you declare a reference and assign it a variable, it will allow you to treat the reference exactly as though it were the original variable for the purpose of accessing and modifying the value of the original variable - even if the second name (the reference) is located within a different scope.

Basic Syntax:

Declaring a variable as a reference rather than a normal variable simply entails appending an ampersand to the type name.

```
type &identifier = identifier/constant;
```

When a reference is created, you must tell it which variable it will become an alias for. After you create the reference, whenever you use the variable, you can just treat it as though it were a regular variable. But when you create it, you must initialize it with another variable, whose address it will keep around behind the scenes to allow you to use it to modify that variable.

In a way, this is similar to having a pointer that always points to the same thing. One key difference is that references do not require dereferencing in the same way that pointers do; you just treat them as normal variables. A second difference is that when you create a reference to a variable, you need not do anything special to get the memory address. The compiler figures this out for you.

```
int x;
int &foo = x;

// foo is now a reference to x so this sets x to 56
foo = 56;
std::cout << x <<std::endl;
```

The most common use of references is for function parameters. Reference parameters facilitate the pass-by-reference style of arguments, as opposed to the pass-by-value style. To observe the differences, consider the three swap functions in the program below.

```
void Swap1 (int x, int y)              //    pass-by-value
(objects)
{
        int temp = x;
        x = y;
        y = temp;
}

void Swap2 (int *x, int *y)            //    pass-by-value
(pointers)
{
        int temp = *x;
        *x = *y;
        *y = temp;
}

void Swap3 (int &x, int &y)            //         pass-by-
reference
{
        int temp = x;
        x = y;
        y = temp;
}
```

Annotation:

- Although *Swap1* swaps x and y, this has no effect on the arguments passed to the function, because Swap1 receives a copy of the arguments. What happens to the copy does not affect the original.
- *Swap2* overcomes the problem of *Swap1* by using pointer parameters instead. By dereferencing the pointers, *Swap2* gets to the original values and swaps them.
- *Swap3* overcomes the problem of *Swap1* by using reference parameters instead. The parameters become aliases for the arguments passed to the function and therefore swap them as intended. *Swap3* has the added advantage that its call syntax is the same as *Swap1* and involves no addressing or dereferencing.

3.10 CONSTANTS

Constants are expressions with a fixed value. C++ has two kinds of constants: literal, and symbolic. C++ has two kinds of constants: literal, and symbolic.

3.10.1 Literal constants:

Literal constants are literal numbers used to express particular values within the source code of a program. They are constants because you can't change their values. We have already used these previously to give concrete values to variables or to express messages we wanted our programs to print out, for example, when we wrote:

```
a = 5;
```

the 5 in this piece of code was a literal constant.

Literal constants can be divided in Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values.

Integer Numerals: They are numerical constants that identify *integer decimal values*. Notice that to express a numerical constant we do not have to write quotes (") nor any special character. In addition to decimal numbers (those that all of us are used to use every day) C++ allows the use as literal constants of octal numbers (base 8) and hexadecimal numbers (base 16). If we want to express an octal number we have to precede it with a 0 (zero character). And in order to express a hexadecimal number we have to precede it with the characters 0x (zero, x). For example, the following literal constants are all equivalent to each other:

```
75      // decimal

0113    // octal

0x4b    // hexadecimal
```

Floating-Point Numerals: They express numbers with *decimals* and/or *exponents*. They can include either a decimal point, an e character (that expresses "by ten at the Xth height", where X is an integer value that follows the e character), or both a decimal point and an e character. For example, the following are floating-point literals:

3.14159 // 3.14159

6.02e23 // 6.02 x 10^23

1.6e.19 // 1.6 x 10^.19

3.0 // 3.0

Character and String Literals: There also exist non-numerical constants, like:

'z'

'p'

"Hello world"

"How do you do?"

The first two expressions represent single character constants, and the following two represent string literals composed of several characters. Notice that to represent a single character we enclose it between *single quotes (')* and to express a string (which generally consists of more than one character) we enclose it between *double quotes (")*.

Boolean Literals: There are only two valid Boolean values: *true* and *false*. These can be expressed in C++ as values of type *bool* by using the Boolean literals true and false.

3.10.2 Symbolic constants:

Symbolic constants can be declared in two different ways:

- using the **#define** preprocessor directive, and
- through use of the **const** keyword.

You can define your own names for constants that you use very often without having to resort to memory consuming variables, simply by using the #define preprocessor directive. Its format is:

```
#define identifier value
```

For example:

```
#define PI 3.14159
#define NEWLINE '\n'
```

This defines two new constants: PI and NEWLINE. Once they are defined, you can use them in the rest of the code as if they were any other regular constant, for example:

```
// defined constants: calculate circumference

#include <iostream>
using namespace std;

#define PI 3.14159
#define NEWLINE '\n'

int main ()
{
        double r=5.0;                // radius
        double circumf;

        circumf = 2 * PI * r;
        cout << circumf;
        cout << NEWLINE;

        return 0;
}
```

OUTPUT: 31.4159

In fact the only thing that the compiler preprocessor does when it encounters *#define* directives is to literally replace any occurrence of their identifier (in the previous example, these were PI and NEWLINE) by the code to which they have been defined (3.14159 and '\n' respectively).

There are two major problems with symbolic constants declared using #define. First, because they are resolved by the preprocessor, which replaces the symbolic name with the defined value, #defined symbolic constants do not show up in the debugger. Second, #defined values always have global scope. This means value #defined in one piece of code may have a naming conflict with a value #defined with the same name in another piece of code.

A better way to do symbolic constants is through use of the **const** keyword. Const variables must be assigned a value when declared, and then that value cannot be changed.

```
const double pi = 3.14159
const char newline = '\n'
```

Here, *pi* and *newline* are two typed constants. Although a constant variable might seem like an oxymoron, they can be very useful in helping to document your code. Const variables act exactly like normal variables in every case except that they cannot be assigned to.

3.11 ASSIGNMENT STATEMENTS

The main statement in C++ for carrying out computation and assigning values to variables is the **assignment statement**. For example the following assignment statement:

```
average = (a + b)/2;
```

assigns half the sum of `a` and `b` to the variable `average`. The general form of an assignment statement is:

identifier = *expression* ;

The *expression* is evaluated and then the value is assigned to the *identifier*. It is important to note that the value assigned to *identifier* must be of the same type as *identifier*.

The *expression* can be a single variable, a single constant or involve variables and constants combined by the arithmetic operators. Rounded brackets `()` may also be used in matched pairs in expressions to indicate the order of evaluation.

For example, let us have a look at the following code - I have included the evolution of the content stored in the variables as comments:

```
// assignment operator

#include <iostream>
using namespace std;

int main ()
{
  int a, b;         // a:?,  b:?
  a = 10;           // a:10, b:?
  b = 4;            // a:10, b:4
  a = b;            // a:4,  b:4
  b = 7;            // a:4,  b:7

  cout << "a:";
  cout << a;
  cout << " b:";
  cout << b;

  return 0;
}
```

OUTPUT:

a:4 b:7

This code will give us as result that the value contained in *a* is 4 and the one contained in *b* is 7. Notice how *a* was not affected by the final modification of *b*, even though we declared *a* = *b* earlier (that is because of the right-to-left rule).

The following expression is also valid in C++:

a = b = c = 5;

It assigns 5 to the all the three variables: a, b and c.

3.12 SUMMARY

- A variable is a symbolic name for a memory location in which data can be stored and subsequently recalled.
- A variable has two important attributes - a type and a value
- A variable declaration has the form:
  - type identifier-list;
- Identifiers are names given to variables which distinguish them from all other variables.
- Variables names (identifiers) can only include letters of the alphabet, digits and the underscore character. They must commence with a letter.
- Reserved words are valid identifiers that have special significance to C++.
- C++ is case-sensitive. That is lower-case letters are treated as distinct from upper-case letters.
- Variables can either have global scope or local scope
- All variables and constants that are used in a C++ program must be declared before use. Declaration associates a type and an identifier with a variable.
- C++ allows us to declare a variable anywhere in the program, as also its initialization at run time, using the expressions at the place of declaration.
- Reference variable is the second name for the variable that can be used to read or modify the original data stored in that variable
- Variables can be initialized in two ways:
  - type identifier = initial_value ;
  - type identifier (initial_value) ;
- Literal constants are literal numbers used to express particular values within the source code of a program.
- Literal constants can be divided in Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values.
- Symbolic constants can be declared in two different ways:

- using the #define preprocessor directive, and
- through use of the const keyword.

➢ The assignment statement in C++ is used for carrying out computation and assigning values to variables.

➢ In an assignment statement, the expression on the right hand side of the assignment is evaluated and, if necessary, converted to the type of the variable on the left hand side before the assignment takes place.

3.13 UNIT-END EXERCISES

3.13.1 Questions:

These questions are intended as a self-test for readers.

1) Define variable. State the important attributes of a variable.
2) Define Identifier. State the rules for valid identifiers.
3) Define global and local scope.
4) Explain reference variables with suitable examples.
5) State the different types of constants
6) Which of the following represent valid variable definitions?
    a. int n = -100;
    b. unsigned int i = -100;
    c. signed int = 2.9;
    d. long m = 2, p = 4;
    e. int 2k;
    f. double x = 2 * m;
    g. float y = y * 2;
    h. unsigned double z = 0.0;
    i. double d = 0.67F;
    j. float f = 0.52L;
    k. signed char = -1786;
    l. char c = '$' + 2;
    m. sign char h = '\111';
    n. char *name = "Peter Pan";
    o. unsigned char *num = "276811";
7) Which of the following represent valid identifiers?
    a. identifier
    b. seven_11
    c. _unique_
    d. gross-income
    e. gross$income

f. 2by2
g. default
h. average_weight_of_a_large_pizza
i. variable
j. object.oriented

3.13.2 Programming Projects:

Writing programs that solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied.

1) Given the following definition of a Swap function

```
void Swap (int x, int y)
{
        int temp = x;
        x = y;
        y = temp;
}
```

what will be the value of x and y after the following call:

```
x = 10;
y = 20;
Swap(x, y);
```

2) Assuming that n is 20, what will the following code fragment output when executed?

```
if (n >= 0)
        if (n < 10)
                cout << "n is small\n";
else
        cout << "n is negative\n";
```

3.14 FURTHER READING

Bjarne Stroustrup, "The C++ Programming Language: Second Edition".

Herbert Schildt, "The C++ Complete Reference"

E Balagurusamy, "Object Oriented Programming C++", Third Edition.

4

Data Types and Expressions

Unit Structure:

4.1 OBJECTIVES

After completing this chapter you will be able to:

- Understand and Identify different data types used in C++
- Understand various operators
- Understand expressions and utilise them in programs

4.2 INTRODUCTION

The memory in our computers is organized in bytes. A byte is the minimum amount of memory that we can manage in C++. A byte can store a relatively small amount of data: one single character or a small integer (generally an integer between 0 and 255). In addition, the computer can manipulate more complex data types that come from grouping several bytes, such as long numbers or non-integer numbers.

When programming, we store the variables in our computer's memory, but the computer has to know what kind of data we want to store in them, since it is not going to occupy the same amount of memory to store a simple number than to store a single letter or a large number, and they are not going to be interpreted the same way.

This chapter simply provides the most basic elements from which C++ programs are constructed. You must know these elements, plus the terminology and simple syntax that goes with them, in order to complete a real project in C++ and especially to read code written by others.

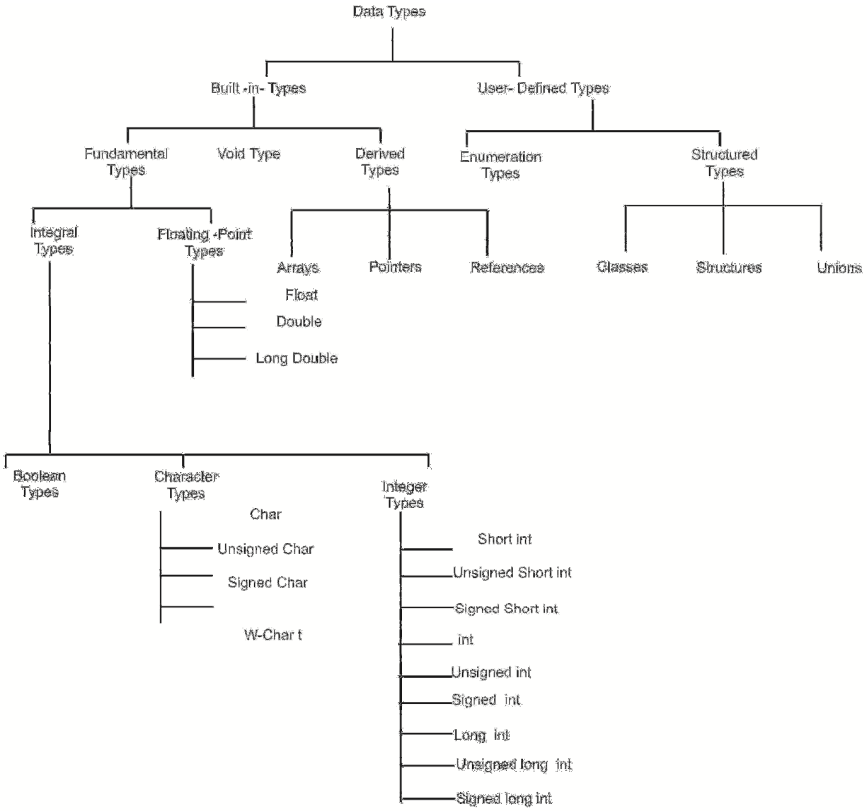
4.3 DATA TYPES

Every name (identifier) in a C++ program has a type associated with it. This type determines what operations can be applied to the name (that is, to the entity referred to by the name) and how such operations are interpreted. For example, the declarations

```
float x;          //x is a floating-point variable
int y = 7;        //y is an integer variable with
the initial value 7
float f(int);     //f is a function taking an
argument of type int and returning a floating-point
number
```

would make the example meaningful. Because y is declared to be an int, it can be assigned to, used in arithmetic expressions, etc. On the other hand, f is declared to be a function that takes an int as its argument, so it can be called given a suitable argument.

Data types in Standard C++ are classified as shown in the diagram below.



The *Boolean*, *character*, and *integer* types are collectively called *integral* types. The *integral* and *floating-point* types are collectively called *arithmetic* types. *Enumeration* and *structured* types are called *user-defined* types because they must be defined by users rather than being available for use without previous declaration, the way fundamental types are. In contrast, other types are called *built-in* types.

The *integral* and *floating-point* types are provided in a variety of sizes to give the programmer a choice of the amount of storage consumed, the precision, and the range available for computations. The assumption is that a computer provides bytes for holding characters and words, for holding and computing - integer values, some entity most suitable for floating-point computation, and addresses for referring to those entities. The C++ fundamental types together with *pointers* and *arrays* present these machine-level notions to the programmer in a reasonably implementation independent manner.

For most applications, one could simply use *bool* for logical values, *char* for characters, *int* for integer values, and *double* for floating-point values. The remaining fundamental types are variations for optimizations and special needs that are best ignored until such needs arise. They must be known, however, to read old C and C++ code.

4.3.1 Booleans:

A Boolean, *bool*, can have one of the two values *true* or *false*. A Boolean is used to express the results of logical operations. For example:

```
void f(int a, int b)
{
      bool b1 = a == b; /  / = is assignment, == is
equality
```

```
        / / ...
}
```

If a and b have the same value, b1 becomes true; otherwise, b1 becomes false.

A common use of bool is as the type of the result of a function that tests some condition (a predicate). For example:

```
bool is_open (File*);
bool greater(int a, int b)  { return a>b;  }
```

By definition, true has the value 1 when converted to an integer and false has the value 0. Conversely, integers can be implicitly converted to bool values: nonzero integers convert to true and 0 converts to false. For example:

```
bool b = 7;        // bool(7) is true, so b
becomes true
int i = true;     // int(true) is 1, so i becomes 1
```

In arithmetic and logical expressions, bools are converted to ints; integer arithmetic and logical operations are performed on the converted values. If the result is converted back to bool, a 0 is converted to false and a nonzero value is converted to true.

```
void g()
{
      bool a = true;
      bool b = true;
      bool x = a + b;          // a+b is 2, so x
      becomes true
      bool y = a - b;          // a-b is 0, so y
      becomes false
}
```

4.3.2 Character Types:

A variable of type char can hold a character of the implementation's character set. For example:

```
char ch = 'a';
```

Almost universally, a char has 8 bits so that it can hold one of 256 different values. Typically, the character set is a variant of ISO-646, for example ASCII, thus providing the characters appearing on your keyboard. Each character constant has an integer value. For example, the value of 'b' is 98 in the ASCII character set. Here is a small program that will tell you the integer value of any character you care to input:

```
#include <iostream>
int main()
{
      char c;
      std::cin >> c;
      std::cout << "The value of '" << c << "' is" <<
int(c) << '\n';
}
```

The notation ***int***(c) gives the integer value for a character *c*. The possibility of converting a char to an integer raises the question: is a char signed or unsigned? The 256 values represented by an 8-bit byte can be interpreted as the values 0 to 255 or as the values -127 to 127. Unfortunately, which choice is made for a plain char is implementation-defined. C++ provides two types for which the answer is definite; signed char, which can hold at least the values -127 to 127, and unsigned char, which can hold at least the values 0 to 255.

Fortunately, the difference matters only for values outside the 0 to 127 range, and the most common characters are within that range.

A type *wchar_t* is provided to hold characters of a larger character set such as Unicode. It is a distinct type. The size of *wchar_t* is implementation-defined and large enough to hold the largest character set supported by the implementation's locale. The strange name is a leftover from C. In C, *wchar_t* is a *typedef* rather than a built-in type. The suffix *'_t'* was added to distinguish standard *typedefs*.

Note that the character types are integral types so that arithmetic and logical operations apply.

4.3.3 Integer Types:

Like char, each integer type comes in three forms: *"plain" int, signed int, and unsigned int*. In addition, integers come in three sizes: *short int, "plain" int, and long int*. A *long int* can be referred to as plain long. Similarly, short is a synonym for *short int*, unsigned for *unsigned int*, and signed for *signed int*. The unsigned integer types are ideal for uses that treat storage as a bit array. Using an unsigned instead of an int to gain one more bit to represent positive integers is almost never a good idea. Attempts to ensure that some values are positive by declaring variables unsigned will typically be defeated by the implicit conversion rules. Unlike plain chars, plain ints are always signed. The signed int types are simply more explicit synonyms for their plain int counterparts.

4.3.4 Floating-Point Types:

The floating-point types represent floating-point numbers. Like integers, floating-point types come in three sizes: *float* (single-precision), *double* (double-precision), and *long double* (extended-precision).

The exact meaning of single-, double-, and extended-precision is implementation-defined. Choosing the right precision for a problem where the choice matters requires significant understanding of floating-point computation. If you don't have that understanding, get advice, take the time to learn, or use *double* and hope for the best.

4.3.5 Sizes:

Some of the aspects of C++'s fundamental types, such as the size of an ***int***, are implementation - defined. I point out these dependencies and often recommend avoiding them or taking steps to minimize their impact. Why should you bother? People who program on a variety of systems or use a variety of compilers care a lot because if they don't, they are forced to waste time finding and fixing obscure bugs. If your program is a success, it is likely to be ported, so someone will have to find and fix problems related to implementation-dependent features. In addition, programs often need to be compiled with other compilers for the same system, and even a future release of your favourite compiler may do some things differently from the current one. It is far easier to know and limit the impact of implementation dependencies when a program is written than to try to untangle the mess afterwards.

It is relatively easy to limit the impact of implementation-dependent language features. Limiting the impact of system-

dependent library facilities is far harder. Using standard library facilities wherever feasible is one approach.

Sizes of C++ objects are expressed in terms of multiples of the size of a char, so by definition the size of a char is 1. The size of an object or type can be obtained using the ***sizeof*** operator. Following is a complete C++ program showing the number of bytes that the fundamental types occupy in the memory.

```
#include <iostream>
using namespace std;
int main()
{
      cout << "size of char         ="          <<
sizeof(char)      << endl;
      cout << "size of short        ="          <<
sizeof(short)     << endl;
      cout << "size of int          ="          <<
sizeof(int)       << endl;
      cout << "size of long         ="          <<
sizeof(long)      << endl;
      cout << "size of float        ="          <<
sizeof(float)     << endl;
      cout << "size of double       ="          <<
sizeof(char)      << endl;
}
```

The ***char*** type is supposed to be chosen by the implementation to be the most suitable type for holding and manipulating characters on a given computer; it is typically an 8-bit byte. Similarly, the ***int*** type is supposed to be chosen to be the most suitable for holding and manipulating integers on a given computer; it is typically a 4-byte (32-bit) word. It is unwise to assume more. For example, there are machines with 32 bit chars.

Next you have a summary of the basic fundamental data types in C++, as well as the range of values that can be represented with each one:

Name	Size	Range
Char	1 byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	2 bytes	signed: -32768 to 32767 unsigned: 0 to 65535
Int	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
Bool	1 byte	true or false
Float	4 bytes	+/- 3.4e +/- 38 (~7 digits)
Double	8 bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8 bytes	+/- 1.7e +/- 308 (~15 digits
wchar_t	2 or 4 bytes	1 wide character

4.3.6 Void:

The type *void* is syntactically a fundamental type. It can, however, be used only as part of a more complicated type; there are no objects of type void. It is used either to specify that a function does not return a value or as the base type for pointers to objects of unknown type. For example:

```
void x;     // error: there are no void objects
```

```
void f()  ; // function f does not return a value
void *pv;   // pointer to object of unknown type
```

When declaring a function, you must specify the type of the value returned. Logically, you would expect to be able to indicate that a function didn't return a value by omitting the return type. However, that would make the grammar less regular and clash with C usage. Consequently, *void* is used as a ''pseudo return type'' to indicate that a function doesn't return a value.

4.3.7 Enumerations:

An ***enumeration*** is a type that can hold a set of values specified by the user. Once defined, an *enumeration* is used very much like an integer type. Named integer constants can be defined as members of an enumeration. For example,

```
enum { ASM, AUTO, BREAK };
```

defines three integer constants, called enumerators, and assigns values to them. By default, enumerator values are assigned increasing from 0, so ASM = 0, AUTO = 1, and BREAK = 2. An enumeration can be named. For example:

```
enum keyword { ASM, AUTO, BREAK };
```

Each *enumeration* is a distinct type. The type of an enumerator is its enumeration. For example,
AUTO is of type keyword.

Declaring a variable *keyword* instead of *plain int* can give both the user and the compiler a hint as to the intended use. For example:

```
void f(keyword key)
{
      switch (key)
      {
            case ASM:
                  // do something
                  break;
            case BREAK:
                  // do something
                  break;
      }
}
```

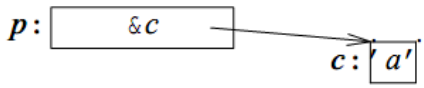
A compiler can issue a warning because only two out of three keyword values are handled. By default, enumerations are converted to integers for arithmetic operations. An *enumeration* is a *user-defined type*, so users can define their own operations, such as ++ and << for an enumeration.

4.3.8 Pointers:

For a type T, T* is the type "***pointer*** to T". That is, a variable of type T* can hold the address of an object of type T. For example:

```
char c =  'a';
char *p = &c;           // p holds the address of c
```

or graphically:



Unfortunately, pointers to arrays and pointers to functions need a more complicated notation:

```
int *pi;          // pointer to int
char **ppc;       // pointer to pointer to char
int *ap[15];      // array of 15 pointers to ints
```

```
int (*fp)(char*); // pointer to function taking a
char* argument;
                  returns an int
int *f(char*);    //  function  taking  a  char*
argument; returns a
                  pointer to int
```

The fundamental operation on a ***pointer*** is dereferencing, that is, referring to the object pointed to by the pointer. This operation is also called indirection. The dereferencing operator is (prefix) unary *. For example:

```
char c = 'a';
char *p = &c;          // p holds the address of c
char c2 = *p;          // c2 == 'a'
```

The variable pointed to by p is c, and the value stored in c is 'a', so the value of *p assigned to c2 is 'a'.

It is possible to perform some arithmetic operations on *pointers* to array elements. *Pointers* to functions can be extremely useful. The implementation of pointers is intended to map directly to the addressing mechanisms of the machine on which the program runs.

4.3.9 Arrays:

For a type T, T[size] is the type "***array*** of size elements of type T". The elements are indexed
from 0 to size - 1. For example:

```
float v[3];       //  an  array  of  three  floats:
v[0], v[1], v[2]
char *a[32];           // an array of 32 pointers
to char: a[0] .. a[31]
```

The number of elements of the array, the array bound, must be a constant expression. If you need variable bounds, use a ***vector***. For example:

```
void f(int i)
{
      int v1[i];  //  error:  array  size  not  a
      constant expression
      vector <int> v2 (i);    // ok
}
```

Multidimensional arrays are represented as arrays of arrays. For example:

```
int d2[10][20];         //  d2  is  an  array  of  10
arrays of 20 integers
```

4.3.10 References:

A ***reference*** is an alternative name for an object. The main use of references is for specifying arguments and return values for functions in general and for overloaded operators in particular. The notation X& means reference to X. To ensure that a reference is a name for something (that is, bound to an object), we must initialize the reference. For example:

```
int i = 1;
int &r1 = i;           // ok: r1 initialized
int &r2;          // error: initializer missing
extern int &r3;         //   ok:   r3   initialized
elsewhere
```

References to variables and references to constants are distinguished because the introduction of a temporary in the case of the variable is highly error-prone; an assignment to the variable would become an assignment to the – soon to disappear – temporary. No such problem exists for references to constants, and references to

constants are often important as function arguments. A reference can be used to specify a function argument so that the function can change the value of an object passed to it.

4.3.11 Structures:

An array is an aggregate of elements of the same type. A ***struct*** is an aggregate of elements of (nearly) arbitrary types. For example:

```
struct address
{
      char *name;
      long int number;
      char * street;
      char * town;
      char state[2];
      long zip;
 };
```

This defines a new type called address consisting of the items you need in order to send mail to someone. Note the semicolon at the end. This is one of very few places in C++ where it is necessary to have a semicolon after a curly brace, so people are prone to forget it. Variables of type address can be declared exactly as other variables, and the individual members can be accessed using the **.** ***(dot)*** operator. For example:

```
void f()
{
      address jd;
      jd.name = "Pratap";
      jd.number = 11;
}
```

Structure objects are often accessed through pointers using the −> (structure pointer dereference) operator. For example:

```
void print_addr(address * p)
{
      cout  << p->name << '\n'
            << p->number << ' ' << p->street << '\n'
            << p->town << '\n'
            << p->state[0] << p->state[1] << '' << p->zip << '\n';
}
```

When p is a pointer, p −> m is equivalent to (*p).m.

Objects of structure types can be assigned, passed as function arguments, and returned as the result from a function. For example:

```
address current;
address set_current (address next)
{
      address prev = current;
      current = next;
      return prev;
}
```

Other plausible operations, such as comparison (== and !=), are not defined. However, the user can define such operators.

A ***struct*** is a simple form of a ***class***.

4.4 OPERATORS

This section introduces the built-in C++ operators for composing expressions. C++ provides operators for composing arithmetic, relational, logical, bitwise, and conditional expressions. It also provides operators which produce useful side-effects, such as

assignment, increment, and decrement. We will look at each category of operators in turn and also discuss the precedence rules which govern the order of operator evaluation in a multi - operator expression.

4.4.1 Arithmetic Operators:

C++ provides five basic *arithmetic operators*. These are summarized in the table below.

Operator	Name
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo (Remainder)

Except for modulo (%) all other arithmetic operators can accept a mix of integer and real operands. Generally, if both operands are integers then the result will be an integer. However, if one or both of the operands are reals then the result will be a real (or double to be exact).

Operations of addition, subtraction, multiplication and division literally correspond with their respective mathematical operators. The only one that you might not be so used to see is modulo; whose operator is the percentage sign (%). Modulo is the operation that gives the remainder of a division of two values. The remainder operator (%) expects integers for both of its operands. It returns the remainder of integer-dividing the operands. For example 13%3 is calculated by integer dividing 13 by 3 to give an outcome of 4 and a remainder of 1; the result is therefore 1.

4.4.2 Relational Operators:

C++ provides six *relational operators* for comparing numeric quantities. These are summarized in the table below. Relational operators evaluate to 1 (representing the true outcome) or 0 (representing the false outcome).

Operator	Name	Example
==	Equality	5 == 5 //gives 1
!=	Inequality	5 != 5 //gives 0
<	Less Than	5 < 5.5 //gives 1
<=	Less Than or Equal	5 <= 5 //gives 1
>	Greater Than	5 > 5.5 //gives 0
>=	Greater Than or Equal	5 >= 6 //gives 0

Note that the <= and >= operators are only supported in the form shown. In particular, =< and => are both invalid and do not mean anything.

The relational operators should not be used for comparing strings, because this will result in the string addresses being compared, not the string contents. For example, the expression

"HELLO" < "BYE"

causes the address of "HELLO" to be compared to the address of "BYE". As these addresses are determined by the compiler (in a machine-dependent manner), the outcome may be 0 or may be 1, and is therefore undefined. C++ provides library functions (e.g., ***strcmp***) for the lexicographic comparison of string. These will be described later in the book.

4.4.3 Logical Operators:

C++ provides three *logical operators* for combining logical expression. These are summarized in the table below. Like the relational operators, logical operators evaluate to 1 or 0.

Operator	Name
!	Logical Negation
&&	Logical And
||	Logical Or

Logical ***negation*** is a unary operator, which negates the logical value of its single operand. If its operand is nonzero it produces 0, and if it is 0 it produces 1. Logical ***and*** produces 0 if one or both of its operands evaluate to 0. Otherwise, it produces 1. Logical ***or*** produces 0 if both of its operands evaluate to 0. Otherwise, it produces 1.

The following panels show the result of Logical And and Or operators evaluating two expressions ‘a’ and ‘b’:

a	b	a && b
True	True	True
True	False	False
False	True	False
False	False	False

a	b	a || b
True	True	True
True	False	True
False	True	True
False	False	False

4.4.4 Bitwise Operators:

C++ provides six *bitwise operators* for manipulating the individual bits in an integer quantity. These are summarized in the table below.

Operator	Name
~	Bitwise Negation
&	Bitwise And
|	Bitwise Or
^	Bitwise Exclusive Or
<<	Bitwise Left Shift
>>	Bitwise Right Shift

Bitwise operators expect their operands to be integer quantities and treat them as bit sequences. Bitwise **negation** is a unary operator which reverses the bits in its operands. Bitwise **and** compares the corresponding bits of its operands and produces a 1 when both bits are 1, and 0 otherwise. Bitwise **or** compares the corresponding bits of its operands and produces a 0 when both bits are 0, and 1otherwise. Bitwise **exclusive or** compares the corresponding bits of its operands and produces a 0 when both bits are 1 or both bits are 0, and 1 otherwise.

Bitwise **left shift** operator and bitwise **right shift** operator both take a bit sequence as their left operand and a positive integer quantity 'n' as their right operand. The former produces a bit sequence equal to the left operand but which has been shifted 'n' bit positions to the left. The latter produces a bit sequence equal to the left operand but which has been shifted 'n' bit positions to the right. Vacated bits at either end are set to 0.

4.4.5 Increment/Decrement Operators:

The *auto increment (++)* and *auto decrement (--)* operators provide a convenient way of, respectively, adding and subtracting 1 from a numeric variable. These are summarized in the table below. The examples assume the following variable definition:

```
int k = 5;
```

Operator	Name	Example
++	Auto Increment (prefix)	++k + 10 //gives 16
++	Auto Increment (postfix)	k++ + 10 //gives 15
--	Auto Decrement (prefix)	--k + 10 //gives 14
--	Auto Decrement (postfix)	k-- + 10 //gives 15

Both operators can be used in prefix and postfix form. The difference is significant. When used in prefix form, the operator is first applied and the outcome is then used in the expression. When used in the postfix form, the expression is evaluated first and then the operator applied.

4.4.6 Assignment Operator:

The *assignment operator* is used for storing a value at some memory location (typically denoted by a variable). Its left operand should be an lvalue, and its right operand may be an arbitrary expression. The latter is evaluated and the outcome is stored in the location denoted by the lvalue. An lvalue (standing for left value) is anything that denotes a memory location in which a value may be stored. The kind of lvalue we have seen so far is a variable, pointers and references.

The assignment operator has a number of variants, obtained by combining it with the arithmetic and bitwise operators. These are summarized in the table below. The examples assume that ***n*** is an integer variable.

Operator	Example	Equivalent to
=	n = 25	
+=	n += 25	n = n+25
-=	n -= 25	n = n - 25
*=	n *= 25	n = n * 25
/=	n /= 25	n = n / 25
%=	n %= 25	n = n % 25
&=	n &= 0xF2F2	n = n & 0xF2F2
|=	n |= 0xF2F2	n = n | 0xF2F2
^=	n ^= 0xF2F2	n = n ^ 0xF2F2
<<=	n <<= 0xF2F2	n = n << 0xF2F2

>>=	n >>= 0xF2F2	n = n >> 0xF2F2

4.4.7 Conditional Operator:

The *conditional operator* takes three operands. It has the general form:

```
operand1 ? operand2 : operand3
```

First operand1 is evaluated, which is treated as a logical condition. If the result is nonzero then operand2 is evaluated and its value is the final result. Otherwise, operand3 is evaluated and its value is the final result. For example:

```
int m = 1, n = 2;
int min = (m < n ? m : n); // min receives 1
```

Note that of the second and the third operands of the conditional operator only one is evaluated.

4.4.8 Comma Operator:

Multiple expressions can be combined into one expression using the *comma operator*. The comma operator takes two operands. It first evaluates the left operand and then the right operand, and returns the value of the latter as the final outcome. For example:

```
int m, n, min;
int mCount = 0, nCount = 0;
//...
min = (m < n ? mCount++, m : nCount++, n);
```

Here when m is less than n, mCount++ is evaluated and the value of m is stored in min. Otherwise, nCount++ is evaluated and the value of n is stored in min.

4.4.9 Scope Resolution Operator:

C++, like C, is a block – structured language. Blocks and scopes can be used in constructing programs. We know that the same variable can be used to have different meanings in different blocks. In C, the global version of a variable cannot be accessed from within the inner block. C++ resolves this problem by introducing a new operator **::** called the *scope resolution operator*. This can be used to uncover a hidden variable. It takes the following form:

```
:: variable-name
```

This operator allows access to the global version of a variable. For example,

```
int main()
{
      int m = 20;       //m declared, local to main
      {
            int m = 10; //m  declared  again,  local
to inner block
            cout << "m = " << m << "\n";
            cout << ":: m = " << ::m << "\n";
      }
}
```

The output of the above program will be

```
m = 10
m = 20
```

A major application of the scope resolution operator is in the classes to identify the class to which a member function belongs. This will be dealt in detail later when classes are introduced.

4.4.10 Member Dereferencing Operators:

C++ permits us to define a class containing various types of data and functions as members. C++ also permits us to access the class members through pointers. In order to achieve this, C++ provides a set of three pointer – to – member operators. The following table shows these operators and their functions.

Operator	Function
::*	To declare a pointer to a member of a class
*	To access a member using object name and a pointer to that member
->*	To access a member using a pointer to the object and a pointer to that member

Further details on these operators will be meaningful only after we discuss classes, and therefore we defer the use of member dereferencing operators until then.

4.4.11 Memory Management Operators:

C uses **malloc**() and **calloc**() functions to allocate memory dynamically at run time. Similarly, it uses the function **free**() to free dynamically allocated memory. Although C++ supports these functions, it also defines two unary operators **new** and **delete** that perform the task of allocating and freeing the memory in a better and easier way.

An object can be created by using **new**, and destroyed by using **delete**, as and when required. A data object created inside a block with new, will remain in existence until it is explicitly destroyed by using delete. Thus, lifetime of an object is directly under our control and is unrelated to the block structure of the program.

The **new** operator can be used to create objects of any type. It takes the following general form:

```
pointer – variable = new data – type;
```

Examples:

```
p = new int;
q = new float;
```

where p is a pointer of type int and q is a pointer of type float.

When a data object is no longer needed, it is destroyed to release the memory space for reuse. The general form of its use is:

```
delete pointer – variable;
```

For example:

```
delete p;
```

4.4.12 Type Cast Operator:

C++ permits explicit type conversion of variables or expressions using the *type cast operator*. The following two versions are equivalent:

```
(type – name) expression      // C notation
type – name (expression)      // C++ notation
```

Examples:

```
average = sum / (float) i;    //C notation
average = sum / float (i);    //C++ notation
```

ANSI C++ adds the following new cast operators:

- ***const_cast***
- ***static_cast***
- ***dynamic_cast***

- ***reinterpret_cast***

Application of these operators is discussed later.

4.5 OPERATOR PRECEDENCE

When writing complex expressions with several operands, we may have some doubts about which operand is evaluated first and which later. There is an established order with the priority of each operator, and not only the arithmetic ones (those whose preference come from mathematics) but for all the operators which can appear in C++. From ***greatest to lowest priority***, the priority order is as follows:

Level	Operator	Description	Grouping
1	::	scope	Left-to-right
2	() [] . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid	postfix	Left-to-right
3	++ -- ~ ! sizeof new delete	unary (prefix)	Right-to-left
	* &	indirection and reference (pointers)	
	+ -	unary sign operator	
4	(type)	type casting	Right-to-left
5	.* ->*	pointer-to-member	Left-to-right
6	* / %	multiplicative	Left-to-right
7	+ -	additive	Left-to-right
8	<< >>	shift	Left-to-right
9	< > <= >=	relational	Left-to-right
10	== !=	equality	Left-to-right
11	&	bitwise AND	Left-to-right
12	^	bitwise XOR	Left-to-right
13	|	bitwise OR	Left-to-right
14	&&	logical AND	Left-to-right
15	||	logical OR	Left-to-right
16	? :	conditional	Right-to-left
17	= *= /= %= += .= >>= <<= &= ^= |=	assignment	Right-to-left
18	,	comma	Left-to-right

Grouping defines the precedence order in which operators are evaluated in the case that there are several operators of the same level in an expression. All these precedence levels for operators can be manipulated or become more legible by removing possible ambiguities using ***parentheses signs (())***.

4.6 Expressions

An expression is any computation which yields a value. When discussing expressions, we often use the term evaluation. For example, we say that an expression evaluates to a certain value. Usually the final value is the only reason for evaluating the expression. However, in some cases, the expression may also produce side-effects. These are permanent changes in the program state. In this sense, C++ expressions are different from mathematical expressions. Expressions may be of the following seven types:

- Constant expressions
- Integral expressions
- Float expressions
- Pointer expressions
- Relational expressions
- Logical expressions
- Bitwise expressions

An expression may also use combinations of the above expressions. Such expressions are known as compound expressions

4.6.1 Constant Expressions:

Constant expressions consist of only constant values. Examples:

```
15
20 + 5 / 2.0
```

4.6.2 Integral Expressions:

Integral expressions are those which produce integer results after implementing all the automatic and explicit type conversions. Examples:

```
m + n
m * n – 5
5 + int(2.3)
```

where m and n are integer variables.

4.6.3 Float Expressions:

Float expressions are those which, after all conversions, produce floating – point results. Examples:

```
x * y / 10
5 + float(10)
```

where x and y are floating – point variables.

4.6.4 Pointer Expressions:

Pointer expressions produce address values. Examples:

```
&m
ptr + 1
```

where m is a variable and ptr is a pointer.

4.6.5 Relational Expressions:

Relational expressions yield results of type *bool* which takes a value *true* or *false*. Examples:

```
x <= y
a+b == c+d
```

When arithmetic expressions are used on either side of a relational operator, they will be evaluated first and then the results compared. Relational expressions are also known as ***Boolean expressions.***

4.6.6 Logical Expressions:

Logical expressions combine two or more relational expressions and produce *bool* type results. Examples:

```
a>b && x==10
x==10 || y==5
```

4.6.7 Bitwise Expressions:

Bitwise expressions are used to manipulate data at a bit level. They are basically used for testing or shifting bits. Examples:

```
x << 3          // Shift three bit position to
left
y >> 1          // Shift one bit position to
right
```

Shift operators are often used for multiplication and division by powers of two.

4.7 SUMMARY

- Standard C++ provides two different data types – **built-in** types and **user-defined** types.
- The type **`int`** is used for whole numbers which are represented exactly within the computer.
- The type **`float`** is used for real (decimal) numbers. They are held to a limited accuracy within the computer.
- The type **`char`** is used to represent single characters. A **`char`** constant is enclosed in single quotation marks.
- Literal strings can be used in output statements and are represented by enclosing the characters of the string in double quotation marks `"`.
- C++ provides an additional use of **void**, for declaration of generic pointers.
- The **enumerated** data types differ slightly in C++. The tag names of the enumerated data types become new type names. That is, we can declare new variables using these tag names.
- **Pointers** are widely used in C++ for memory management and to achieve polymorphism.
- C++ is very strict regarding type checking of variables. It does not allow to equate variables of two different data types. The only way to break this rule is **type casting**.
- A major application of the **scope resolution (::) operator** is in the classes to identify the class to which a member function belongs.

- C++ provides two new unary operators, in addition to **malloc**(), **calloc**() and **free**() functions, **new** and **delete** to perform the task of allocating and freeing the memory in a better and easier way.

- The order of evaluation of an expression is determined by the precedence of the operators.

- C++ supports seven types of **expressions**. When data types are mixed in an expression, C++ performs the conversion automatically using certain rules.

- When `float` expressions are assigned to `int` variables there may be loss of accuracy.

- C++ also permits **explicit type conversion** of variables and expressions using the type **cast operators**.

4.8 Unit End Exercises

4.8.1 Questions

These questions are intended as a self-test for readers.

1) An unsigned int can be twice as large as the signed int. Explain how?

2) What are the applications of void data type in C++?

3) Describe the implementation of enum data type in C++?

4) In C++, a variable can be declared anywhere in the scope. What is the significance of this feature?

5) What is an operator?

6) What is the application of the scope resolution operator in C++?

7) What data types would you use to represent the following items?
   a. the number of students in a class
   b. the grade (a letter) attained by a student in the class
   c. the average mark in a class
   d. the distance between two points
   e. the population of a city
   f. the weight of a postage stamp
   g. the registration letter of a car

4.8.2 Programming Projects:

Writing programs that solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied.

1) What is wrong with the following code fragment?

```
a. enum Season { SPRING, SUMMER, FALL, WINTER };

b. enum Semester { FALL, SPRING, SUMMER };
```

2) Write declaration statements to declare integer variables ***i*** and ***j*** and float variables ***x*** and ***y***. Extend your declaration statements so that ***i*** and ***j*** are both initialised to 1 and ***y*** is initialised to 10.0.

3) Find errors, if any, in the following C++ statements.

a. long float x;

b. char *cp = vp; //vp is a void pointer

c. int code = three; //three is an enumerator

d. int *p = new; //allocate memory with new

e. enum {green, yellow, red};

f. int const *p = total;

g. const int array_size;

h. int &number = 100;

i. float *p = new int[10];

j. char name[3] = "USA";

4) To what do the following expressions evaluate?
   a. 17/3
   b. 17%3
   c. 1/2
   d. 1/2*(x+y)

5) Given the declarations:

```
float x;
int k, i = 5, j = 2;
```

To what would the variables `x` and `k` be set as a result of the assignments

a. `k = i/j;`
b. `x = i/j;`
c. `k = i%j;`

d. `x = 5.0/j;`

4.9 FURTHER READING

Bjarne Stroustrup, "The C++ Programming Language: Second Edition".

Herbert Schildt, "The C++ Complete Reference"

E Balagurusamy, "Object Oriented Programming C++", Third Edition.
John Hubbard "Fundamentals of Computing with C++"

5

Input and Output

Unit Structure

5.1 OBJECTIVES

After completing this chapter you will be able to:

- Understand the Standard Input and Output stream
- Identify and use escape characters
- Understand Preprocessor directives
- Understand and use namespaces.

5.2 INTRODUCTION

Until now, the example programs of previous sections provided very little interaction with the user, if any at all. Using the standard input and output library, we will be able to interact with the user by printing messages on the screen and getting the user's input from the keyboard.

C++ uses a convenient abstraction called ***streams*** to perform input and output operations in sequential media such as the screen or the keyboard. A ***stream*** is an object where a program can either insert or extract characters to/from it. We do not really need to care about many specifications about the physical media associated with the stream - we only need to know it will accept or provide characters sequentially.

The most common way in which a program communicates with the outside world is through simple, character-oriented input/output (IO) operations. C++ provides two useful operators for this purpose: >> for input and << for output. The standard C++ library includes the header file ***iostream***, where the standard input and output stream objects are declared.

5.3 STANDARD OUTPUT (COUT)

By default, the standard output of a program is the screen, and the C++ stream object defined to access it is ***cout***. ***cout*** is used in conjunction with the ***insertion operator***, which is written as << (two "less than" signs).

```
cout << "Output sentence";     //   prints   Output
sentence on screen
cout << 120;                   // prints number 120
on screen
cout << x;                     // prints the content
of x on screen
```

The << operator inserts the data that follows it into the stream preceding it. In the examples above it inserted the constant string Output sentence, the numerical constant 120 and variable x into the standard output stream cout. Notice that the sentence in the first instruction is enclosed between double quotes (") because it is a constant string of characters. Whenever we want to use constant strings of characters we must enclose them between double quotes (") so that they can be clearly distinguished from variable names. For example, these two sentences have very different results:

```
cout << "Hello";         // prints Hello
cout << Hello;           // prints the content of
Hello variable
```

The insertion operator (<<) may be used more than once in a single statement. The utility of repeating the insertion operator (<<) is demonstrated when we want to print out a combination of variables and constants or more than one variable:

```
cout << "Hello, I am " << age << " years old and my
postal code is " << pincode;
```

If we assume the age variable to contain the value 24 and the pincode variable to contain 400054 the output of the previous statement would be:

```
Hello, I am 24 years old and my postal code is
400054
```

It is important to notice that ***cout*** does not add a line break after its output unless we explicitly indicate it, therefore, the following statements:

```
cout << "This is a sentence.";
cout << "This is another sentence.";
```

will be shown on the screen one following the other without any line break between them:

```
This is a sentence.This is another sentence.
```

even though we had written them in two different insertions into cout. In order to perform a line break on the output we must explicitly insert a new-line character into cout. In C++ a new-line character can be specified as ***\n (backslash, n)***:

```
cout << "First sentence.\n ";
cout << "Second sentence.\nThird sentence.";
```

This produces the following output:

```
First sentence.
Second sentence.
Third sentence.
```

Additionally, to add a new-line, you may also use the ***endl*** manipulator. For example:

```
cout << "First sentence." << endl;
cout << "Second sentence." << endl;
```

would print out:

```
First sentence.
Second sentence.
```

The ***endl*** manipulator produces a newline character, exactly as the insertion of '***\n***' does, but it also has an additional behaviour when it is used with buffered streams: the buffer is flushed. Anyway, cout will be an unbuffered stream in most cases, so you can generally use both the ***\n*** escape character and the ***endl*** manipulator in order to specify a new line without any difference in its behaviour.

5.4 STANDARD INPUT (CIN)

The standard input device is usually the keyboard. Handling the standard input in C++ is done by applying the overloaded operator of extraction (>>) on the ***cin*** stream. The operator must be followed by the variable that will store the data that is going to be extracted from the stream. For example:

```
int age;
cin >> age;
```

The first statement declares a variable of type int called age, and the second one waits for an input from cin (the keyboard) in order to store it in this integer variable.

cin can only process the input from the keyboard once the ***RETURN*** key has been pressed. Therefore, even if you request a single character, the extraction from cin will not process the input until the user presses RETURN after the character has been introduced.

You must always consider the type of the variable that you are using as a container with cin extractions. If you request an integer you will get an integer, if you request a character you will get a character and if you request a string of characters you will get a string of characters.

```
// i/o example
#include <iostream>
using namespace std;
 int main ()
{
      int i;
      cout << "Please enter an integer value: ";
      cin >> i;
      cout << "The value you entered is " << i;
      cout << " and its double is " << i*2 <<
".\n";
      return 0;
}
Output:
Please enter an integer value: 702
The value you entered is 702 and its double is
1404.
```

The user of a program may be one of the factors that generate errors even in the simplest programs that use ***cin*** (like the one we have just seen). Since if you request an integer value and the user introduces a name (which generally is a string of characters), the result may cause your program to misoperate since it is not what we were expecting from the user. So when you use the data input provided by cin extractions you will have to trust that the user of your program will be cooperative and that he/she will not introduce his/her name or something similar when an integer value is requested. You can also use ***cin*** to request more than one datum input from the user:

```
cin >> a >> b;
```

is equivalent to:

```
cin >> a;
cin >> b;
```

In both cases the user must give two data, one for variable **a** and another one for variable **b** that may be separated by any valid blank separator: a space, a tab character or a newline.

5.4.1 cin and Strings:

We can use cin to get strings with the extraction operator (>>) as we do with fundamental data type variables:

```
cin >> mystring;
```

However, as it has been said, cin extraction stops reading as soon as if finds any blank space character, so in this case we will be able to get just one word for each extraction. This behaviour may or may not be what we want; for example if we want to get a sentence from the user, this extraction operation would not be useful. In order to get entire lines, we can use the function ***getline***, which is the more recommendable way to get user input with cin.

```
// cin with strings
```

```
#include <iostream>
#include <string>
using namespace std;
 int main ()
{
      string mystr;
      cout << "What's your name? ";
      getline (cin, mystr);
      cout << "Hello " << mystr << ".\n";
      return 0;
}
Output:
What's your name? Prakash Pratap Singh
Hello Prakash Pratap Singh.
```

Notice how we used the ***getline*** function with ***cin*** to get a complete line using the string identifier (***mystr***).

5.5 ESCAPE CHARACTERS

Escape characters are special characters that are difficult or impossible to express otherwise in the source code of a program, like ***newline*** (***\n***) or ***tab*** (***\t***). All of them are preceded by a ***backslash*** (\). Here you have a list of some of such escape codes:

Name	C++ Name
Newline	\n
Horizontal tab	\t
Vertical tab	\v
Backspace	\b
Carriage return	\r
Form feed	\f
Alert	\a
Backslash	\\
Question mark	\?
Single quote	\’
Double quote	\”
Octal number	\ooo
Hexadecimal number	\xhhh

Despite their appearances, these are single characters.
For example:

```
'\n'
'\t'
"Left \t Right"
"one\ntwo\nthree"
```

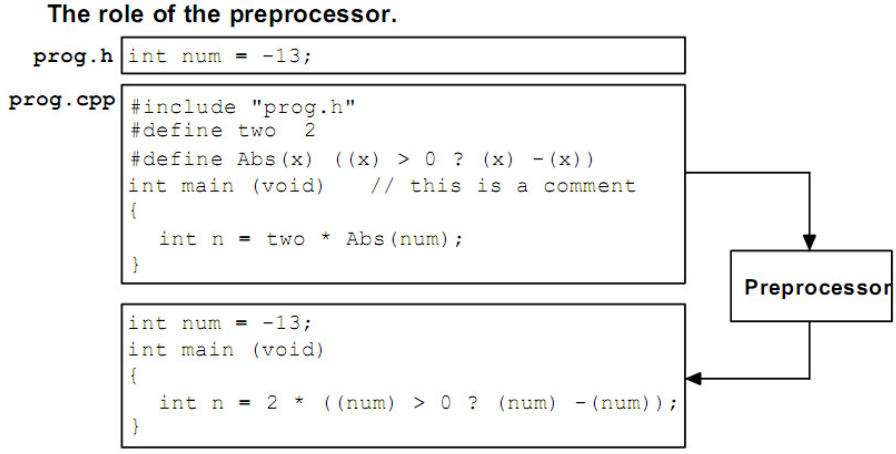
Additionally, you can express any character by its numerical ASCII code by writing a backslash (\) followed by the ASCII code expressed as an octal (base-8) or hexadecimal (base-16) number. In the first case (octal) the digits must immediately follow the backslash (for example \23 or \40), in the second case (hexadecimal), an x character must be written before the digits themselves (for example \x20 or \x4A).

Decimal	Octal	Hexadecimal
6	‘\6’	‘\x6’

48	'\60'	'\x30'
95	'\137'	'\x05F'

5.6 PREPROCESS OR DIRECTIVES

Prior to compiling a program source file, the C++ compiler passes the file through a **preprocessor**. The role of the **preprocessor** is to transform the source file into an equivalent file by performing the preprocessing instructions contained by it. These instructions facilitate a number of features, such as: file inclusion, conditional compilation, and macro substitution. The figure below illustrates the effect of the preprocessor on a simple file.



The **preprocessor** performs very minimal error checking of the preprocessing instructions. Because it operates at a text level, it is unable to check for any sort of language-level syntax errors. This function is performed by the compiler.

The preprocessor is executed before the actual compilation of code begins; therefore the preprocessor digests all these directives before any code is generated by the statements. These preprocessor directives extend only across a single line of code. As soon as a newline character is found, the preprocessor directive is considered to end. No semicolon (;) is expected at the end of a preprocessor directive.

5.6.1 Macro definitions (#define, #undef):

To define preprocessor **macros** we can use **#define**. Its format is:

```
#define identifier replacement
```

When the preprocessor encounters this directive, it replaces any occurrence of ***identifier*** in the rest of the code by ***replacement***. This ***replacement*** can be an expression, a statement, a block or simply anything. The preprocessor does not understand C++; it simply replaces any occurrence of identifier by replacement.

```
#define TABLE_SIZE 100
int table1[TABLE_SIZE];
int table2[TABLE_SIZE];
```

After the preprocessor has replaced TABLE_SIZE, the code becomes equivalent to:

```
int table1[100];
int table2[100];
```

Consider the example given below:

```
// function macro
#include <iostream>
using namespace std;
 #define getmax(a,b) ((a)>(b)?(a):(b))
 int main()
{
        int x=5, y;
        y= getmax(x,2);
        cout << y << endl;
        cout << getmax(7,x) << endl;
        return 0;
}
Output:
5
7
```

#define can work also with parameters to define function macros. Any occurrence of getmax followed by two arguments is replaced by the replacement expression, also replacing each argument by its identifier, exactly as you would expect if it was a function.

Defined macros are not affected by block structure. A macro lasts until it is undefined with the **#undef** preprocessor directive.

```
#define TABLE_SIZE 100
int table1[TABLE_SIZE];
#undef TABLE_SIZE
#define TABLE_SIZE 200
int table2[TABLE_SIZE];
```

This would generate the same code as:

```
int table1[100];
int table2[200];
```

Because preprocessor replacements happen before any C++ syntax check, macro definitions can be a tricky feature, but be careful: code that relies heavily on complicated macros may result obscure to other programmers, since the syntax they expect is on many occasions different from the regular expressions programmers expect in C++.

5.6.2 Conditional inclusions (#ifdef, #ifndef, #if, #endif, #else and #elif):

These directives allow including or discarding part of the code of a program if a certain condition is met. **#ifdef** allows a section of a program to be compiled only if the macro that is specified as the parameter has been defined, no matter which its value is. For example:

```
#ifdef TABLE_SIZE
int table[TABLE_SIZE];
#endif
```

In this case, the line of code int table [TABLE_SIZE]; is only compiled if TABLE_SIZE was previously defined with #define, independently of its value. If it was not defined, that line will not be included in the program compilation.

#ifndef serves for the exact opposite: the code between **#ifndef** and **#endif** directives is only compiled if the specified identifier has not been previously defined. For example:

```
#ifndef TABLE_SIZE
#define TABLE_SIZE 100
#endif
int table[TABLE_SIZE];
```

In this case, if when arriving at this piece of code, the TABLE_SIZE macro has not been defined yet, it would be defined to a value of 100. If it already existed it would keep its previous value since the #define directive would not be executed.

The **#if**, **#else** and **#elif** (i.e., "else if") directives serve to specify some condition to be met in order for the portion of code they surround to be compiled. The condition that follows **#if** or **#elif** can only evaluate constant expressions, including macro expressions. For example:

```
#if TABLE_SIZE>200
      #undef TABLE_SIZE
      #define TABLE_SIZE 200
  #elif TABLE_SIZE<50
      #undef TABLE_SIZE
      #define TABLE_SIZE 50
  #else
      #undef TABLE_SIZE
      #define TABLE_SIZE 100
#endif
int table[TABLE_SIZE];
```

Notice how the whole structure of **#if**, **#elif** and **#else** chained directives ends with **#endif**.

The table below summarizes the general forms of these directives (code denotes zero or more lines of program text, and expression denotes a constant expression).

General Form of Conditional Inclusion Directives:

Form	Explanation

`#ifdef identifier`
`code`
`#endif`
	If identifier is a **#defined** symbol then code is included in the compilation process. Otherwise, it is excluded.

`#ifndef identifier`
`code`
`#endif`
	If identifier is not a **#defined** symbol then code is included in the compilation process. Otherwise, it is excluded.

`#if expression`
`code`
`#endif`
	If expression evaluates to nonzero then code is included in the compilation process. Otherwise, it is excluded.

`#ifdef identifier`
`code1`
`#else`
`code2`
`#endif`
	If identifier is a **#defined** symbol then code1 is included in the compilation process and code2 is excluded. Otherwise, code2 is included and code1 is excluded.
	Similarly, **#else** can be used with **#ifndef** and **#if**.

`#if expression1`	If expression1 evaluates to nonzero then only

`code1`
`#elif expression2`
`code2`
`#else`
`code3`
`#endif`
	code1 is included in the compilation process. Otherwise, if expression2 evaluates to nonzero then only code2 is included. Otherwise, code3 is included.
	As before, the **#else** part is optional. Also, any number of **#elif** directives may appear after **#if** directive.

5.6.3 Source file inclusion (#include):

This directive has been used assiduously in previous sections of this book. When the preprocessor finds an **#include** directive it replaces it by the entire content of the specified file. There are two ways to specify a file to be included:

```
#include "file"
#include <file>
```

The only difference between both expressions is the places (directories) where the compiler is going to look for the file. In the first case where the file name is specified between **double-quotes**, the file is searched first in the same directory that includes the file containing the directive. In case that it is not there, the compiler searches the file in the default directories where it is configured to look for the standard header files.

If the file name is enclosed between **angle-brackets <>** the file is searched directly where the compiler is configured to look for the standard header files. Therefore, standard header files are usually included in angle-brackets, while other specific header files are included using quotes.

5.6.4 Pragma directive (#pragma):

This directive is used to specify diverse options to the compiler. These options are specific for the platform and the compiler you use. Consult the manual or the reference of your compiler for more information on the possible parameters that you can define with **#pragma**. If the compiler does not support a specific argument for **#pragma**, it is ignored - no error is generated.

5.6.5 Error directive (#error):

This directive aborts the compilation process when it is found, generating a compilation the error that can be specified as its parameter:

```
#ifndef __cplusplus
      #error A C++ compiler is required!
#endif
```

This example aborts the compilation process if the macro name **__cplusplus** is not defined (this macro name is defined by default in all C++ compilers).

5.7 NAMESPACES

A namespace is a mechanism for expressing logical grouping. That is, if some declarations logically belong together according to some criteria, they can be put in a common namespace to express that fact. Namespaces allow to group entities like classes, objects and functions under a name. This way the global scope can be divided in "sub-scopes", each one with its own name. The format of namespaces is:

```
namespace identifier
{
      entities
}
```

Where identifier is any valid identifier and entities is the set of classes, objects and functions that are included within the namespace. For example:

```
namespace myNamespace
{
  int a, b;
}
```

In this case, the variables **a** and **b** are normal variables declared within a namespace called myNamespace. In order to access these variables from outside the myNamespace namespace we have to use the scope operator ::. For example, to access the previous variables from outside myNamespace we can write:

```
myNamespace::a
myNamespace::b
```

The functionality of namespaces is especially useful in the case that there is a possibility that a global object or function uses the same identifier as another one, causing redefinition errors. For example:

```
// namespaces
#include <iostream>
using namespace std;
 namespace first
{
      int var = 5;
}
 namespace second
{
      double var = 3.1416;
}
 int main ()
{
      cout << first::var << endl;
      cout << second::var << endl;
      return 0;
}
Output:
5
3.1416
```

In this case, there are two global variables with the same name: **var**. One is defined within the namespace **first** and the other one in **second**. No redefinition errors happen thanks to namespaces.

A **namespace** is a scope. Thus, ‘‘namespace’’ is a very fundamental and relatively simple concept. The larger a program is, the more useful namespaces are to express logical separations of its parts. Ordinary local scopes, global scopes, and classes are namespaces.

Ideally, every entity in a program belongs to some recognizable logical unit (‘‘module’’). Therefore, every declaration in a nontrivial program should ideally be in some namespace named to indicate its logical role in the program. The exception is **main()**, which must be global in order for the run-time environment to recognize it as special.

5.7.1 Using Declaration:

When a name is frequently used outside its namespace, it can be a bother to repeatedly qualify it with its namespace name. The redundancy can be eliminated by a *using-declaration* to state in one place that the **member** used in the scope belongs to a **namespace**. The format of using-declaration is:

```
using name :: member;
```

where *name* refers to the name of the namespace.
For example:

```
// using
#include <iostream>
using namespace std;
namespace first
{
      int x = 5;
      int y = 10;
}
namespace second
{
      double x = 3.1416;
      double y = 2.7183;
}
int main ()
{
      using first::x;
      using second::y;
      cout << x << endl;
      cout << y << endl;
      cout << first::y << endl;
      cout << second::x << endl;
      return 0;
}
Output:
5
2.7183
10
3.1416
```

Notice how in this code, x (without any name qualifier) refers to first::x whereas y refers to second::y, exactly as our using declarations have specified. We still have access to first::y and second::x using their fully qualified names.

A using-declaration introduces a local synonym. It is often a good idea to keep local synonyms as local as possible to avoid confusion.

5.7.2 Using Directive:

The keyword using can also be used as a directive to introduce an entire namespace. A using-directive makes names from a namespace available almost as if they had been declared outside their namespace. The format of using-directive is:

```
using namespace identifier;
```

For example:

```
// using
#include <iostream>
using namespace std;
namespace first
{
      int x = 5;
      int y = 10;
}
namespace second
{
      double x = 3.1416;
double y = 2.7183;
}
int main ()
{
      using namespace first;
      cout << x << endl;
      cout << y << endl;
      cout << second::x << endl;
      cout << second::y << endl;
      return 0;
}
Output:
5
10
3.1416
2.7183
```

In this case, since we have declared that we were using ***namespace first***, all direct uses of **x** and **y** without name qualifiers was referring to their declarations in namespace **first**.

Global using-directives are a tool for transition and are otherwise best avoided. In a namespace, a using-directive is a tool for namespace composition. In a function (only), a using-directive can be safely used as a notational convenience.

using (*using-declaration*) and ***using namespace*** (*using-directive*) have validity only in the same block in which they are stated or in the entire code if they are used directly in the global scope.

5.7.3 Namespace std:

All the files in the C++ standard library declare all of its entities within the ***std*** namespace. That is why we have generally included the using namespace ***std***; statement in all programs that used any entity defined in ***iostream***.

5.8 COMMENTS AND INDENTATION

A comment is a piece of descriptive text which explains some aspect of a program. Program comments are totally ignored by the compiler and are only intended for human readers. C++ provides two types of comment delimiters:

- Anything after // (until the end of the line on which it appears) is considered a comment.

- Anything enclosed by the pair /* and */ is considered a comment.

The following program illustrates the use of both forms:

```
#include <iostream.h>
/* This program calculates the weekly gross pay for
a worker based on the total number of hours worked
and the hourly pay rate. */
int main (void)
{
      int workDays = 5; // Number of work days per
      week
      float workHours = 7.5; // Number of work
      hours per day
      float payRate = 33.50; // Hourly pay rate
      float weeklyPay; // Gross weekly pay
      weeklyPay = workDays * workHours * payRate;
      cout << "Weekly Pay = " << weeklyPay << '\n';
}
```

Judicious use of comments and consistent use of indentation can make the task of reading and understanding a program much more pleasant. Several different consistent styles of indentation are in use. I see no fundamental reason to prefer one over another (although, like most programmers, I have my preferences). The same applies to styles of comments.

Comments should be used to enhance (not to hinder) the readability of a program. The following points, in particular, should be noted:

- A comment should be easier to read and understand than the code which it tries to explain. A confusing or unnecessarily-complex comment is worse than no comment at all.

- Over-use of comments can lead to even less readability. A program which contains so much comment that you can hardly see the code can by no means be considered readable.

- Use of descriptive names for variables and other entities in a program, and proper indentation of the code can reduce the need for using comments.

Once something has been stated clearly in the language, it should not be mentioned a second time in a comment. For example:

```
a = b+c;     // a becomes b+c
count++;     // increment the counter
```

Such comments are worse than simply redundant. They increase the amount of text the reader has to look at, they often obscure

the structure of the program, and they may be wrong. Note, however, that such comments are used extensively for teaching purposes in programming language textbooks such as this. This is one of the many ways a program in a textbook differs from a real program.

My preference is for:

1) A comment for each source file stating what the declarations in it have in common, references to manuals, general hints for maintenance, etc.

2) A comment for each class, template, and namespace

3) A comment for each nontrivial function stating its purpose, the algorithm used (unless it is obvious), and maybe something about the assumptions it makes about its environment

4) A comment for each global and namespace variable and constant

5) A few comments where the code is non-obvious and/or non-portable

6) Very little else.

A well-chosen and well-written set of comments is an essential part of a good program. Writing good comments can be as difficult as writing the program itself. It is an art well worth cultivating.

5.9 SUMMARY

- The standard output in C++ is done by applying the overloaded operator of insertion (<<) on the **cout** stream.
- The standard input in C++ is done by applying the overloaded operator of extraction (>>) on the **cin** stream.
- **Escape** characters are special characters preceded by a backslash (\).
- **Preprocessor directives** are lines included in the code of our programs that are not program statements but directives for the preprocessor. These lines are always preceded by a hash sign (#).
- Preprocessor directives are
  - Macro definitions - #define, and #undef
  - Conditional Inclusions - #ifdef, #ifndef, #if, #endif, #else, and #elif
  - Source File Inclusion - #include
  - Pragma Directive - #pragma
  - Error Directive - #error

- When you have a choice, prefer the standard library to other libraries.
- Do not think that the standard library is ideal for everything.
- Remember to **#include** the headers for the facilities you use.
- Remember that standard library facilities are defined in namespace **std**.
- Use **namespaces** to express logical structure.
- Place every nonlocal name, except main(), in some namespace.
- Use the **Namespace :: member** notation when defining namespace members.
- Use **using namespace** only for transition or within a local scope.
- Keep **comments** crisp.
- Maintain a consistent **indentation** style.

5.10 UNIT END EXERCISES

5.10.1 Questions:

These questions are intended as a self-test for readers.

1) Explain role of preprocessor?
2) What is a preprocessor directive? Explain them?
3) What is a namespace? Explain its functionality?
4) How does using-declaration differ from using-directive?
5) What is wrong with these macro definitions?
    a. #define PI = 3.141593;
    b. #define MAX(a,b) a>b ? a : b
    c. #define fac(a) (a)*fac((a)-1)
6) Write directives for the following:
    a. Defining **Small** as an **unsigned char** when the symbol **PC** is defined, and as **unsigned short** otherwise.
    b. Including the file **basics.h** in another file when the symbol **CPP** is not defined.
    c. Including the file **debug.h** in another file when **release** is 0, or **beta.h** when **release** is 1, or **final.h** when **release** is greater than 1.

5.10.2 Programming Projects:

1) Writing programs that solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied.

2) Write a program like ''Hello, world!'' that takes a name as a command-line argument and writes ''Hello, name!''. Modify this program to take any number of names as arguments and to say hello to each.

3) What will the following program output when executed?

```
#include <iostream.h>
char *str = "global";
void Print (char *str)
{
        cout << str << '\n';
        {
             char *str = "local";
             cout << str << '\n';
             cout << ::str << '\n';
        }
        cout << str << '\n';
}
int main (void)
{
        Print("Parameter");
        return 0;
}
```

5.11 FURTHER READING

Bjarne Stroustrup, "The C++ Programming Language: Second Edition".

Herbert Schildt, "The C++ Complete Reference"

E Balagurusamy, "Object Oriented Programming C++", Third Edition.
John Hubbard "Fundamentals of Computing with C++"

6

Chapter 6

Decisions

Unit Structure

6.1. Introduction

The execution of statements in a program by default is sequential. But sometimes there are such situations where the problem statement and the program logic demands that the program execution or flow be directed or branched to a particular set of statements rather than the other. Ex. Finding greater of 2 given numbers.

This kind of situation involves decision making. C++ offers the following decision control structures:

1. if
2. if-else
3. conditional operator
4. switch.

6.2 Decision Control Structure

6.2.1 if statement:

The syntax of if statement is as follows:

```
if (expression is true)
      execute  statement;
```

- Here, if is a keyword. It tells the compiler that what follows is a decision control structure.
- An if statement is always followed by an expression or condition which is enclosed within a pair of parenthesis.
- The expression is evaluated and will be either true or false. If true then the statement following the if statement is executed, if false then this statement is skipped and the execution continues from next statement.
- Every non zero value will be considered true be it positive or negative.
- Expression or condition is usually a combination of variables and or constants and operators.
- Examples of expressions used in if statement:

  (a>b) // combination of variables & relational operator
  (x<5) // combination of variables & relational operator

- In general an expression is formed using relational operators. Relational operators are used to compare the values of two variables. We can use the relational operators to construct the following types of expressions:

Expression	Is true if
A >B	A is greater than B
A < B	A is less than B
A <= B	A is less than or equal to B
A >= B	A is greater than or equal to B
A == B	A is equal to B
A != B	A is not equal to B

- Example : The following program written in c++ prompts the user for a number. If the user enters the number 3 it prints a particular string, if the user enters any other number it prints nothing.

```
/*************************************************
*******
Program 6.1.
Author : Nikhil Pawanikar
Description : Program to take one input from user. If number
equals to 3
```

```
                Print a string on the screen, if not do nothing.
**************************************************
*******/
#include<iostream.h>
#include<conio.h>
int main()
{
int a;
cout<<"enter the value of a\t";
cin>>a;
if(a==3)
        cout<<"A equals to 3";
getch();
return 0;
}
```

- **Output**

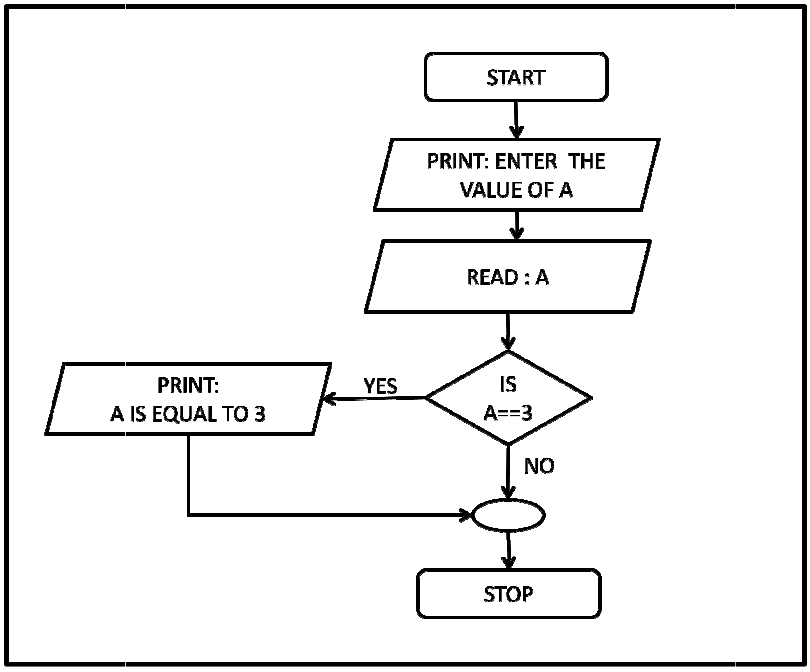
First Run:

```
enter the value of a        4        //nothing happens
```

Second Run:

```
enter the value of a        3
A equals to 3
```

- The above program and the execution of if statement will be better understood by the following flowchart:



- In case multiple statements are to be executed if the expression is true, those statements can be put inside a set of parenthesis { } . If the condition is true the statements in the block following the parenthesis is executed.
- Ex.

```
if(a<b)
{
cout<<"A is less than b";
a=a+1;
}
```

Note: A semicolon is not used with if statement after the } end bracket, it is used only with simple if statement that executes single instruction.

6.2.2 if-else statement:

- The general syntax of if – else statement is as follows

```
if (expression is true)
                execute  statement;
else
                execute statement;
```

Note: single statement is executed if condition is true else single statement is excuted if condition is false

OR

```
if (expression is true)
{
        execute statement;
        execute statement;
        execute statement;
}
else
{
        execute statement;
        execute statement;
        execute statement;
}
```

Note: The Block of statements in parenthesis {} is executed if condition is true else the other block of statements following else is executed if condition is false.

- The if-else statement is slightly different version of if statement. Here if the condition is found to be false other statements are executed. If the condition or expression evaluates to true the statement (single)/ block of statements following if is executed, if it evaluates to false then the statement(single)/block of statement s following the keyword else is executed.
- The expressions/conditions are the same as described above in if statement and may be relational expressions.
- Example : Consider the following program that prompts the user to enter 2 numbers and prints the greater of two numbers on the screen.

```
/*************************************************
*******
Program 6.2
Author : Nikhil Pawanikar
Description : Program to take two inputs from user and printe
the greater
                Of the two
*************************************************
*******/

#include<iostream.h>
#include<conio.h>
```

```
int main()
{
int a,b;
cout<<"enter the value of a & b\t";
cin>>a>>b;
if(a>b)
        cout<<"\n A is greater than B";
else
        cout<<"\n B is greater than A";
getch();
return 0;
}
```

- Output

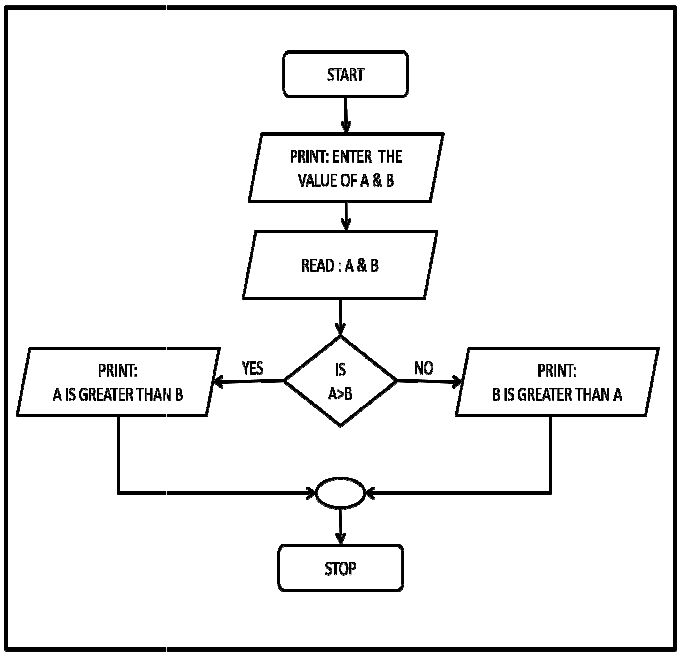
First Run:

```
Enter the value of a & b        2  4
B is greater than A
```

Second Run:

```
Enter the value of a & b        8  4
A is greater than B
```

- The above program could be better understood with the following flowchart:



Nested if-else statements:

- C++ allows nesting of if-else statements ie. We can have another if or if-else statement inside either the if part or else part as follows:

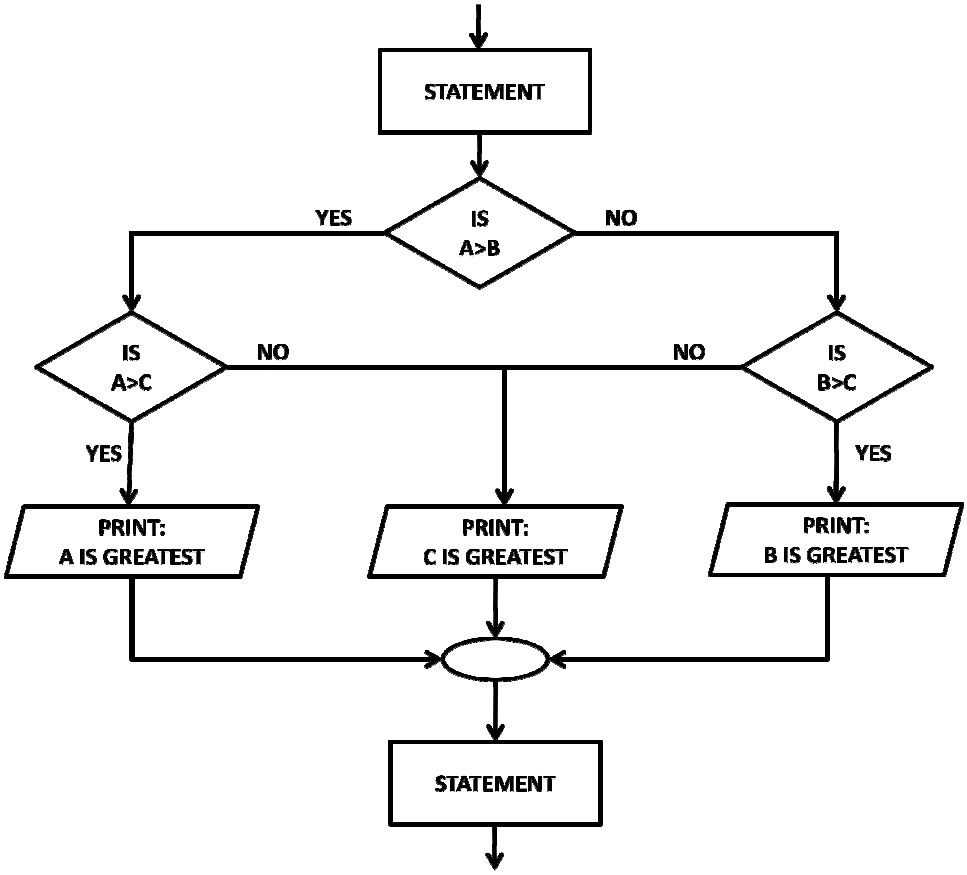
```
if(condition)
 {
        statements;
        if(condition)
         {

        statements;
         }
        else
         {

        statements;
         }
         statements;
 }
else
 {
        statements;
 }
```

```
if(condition)
 {
        statements;
 }
else
 {
        statements;
        if(condition)
         {

        statements;
         }
        else
         {

        statements;
         }
        statements;
 }
```

```
/*************************************************************
****
Program 6.3
Author : Nikhil Pawanikar
Description : Program to take three inputs from user and print the
greater
                Of the three using nested if-else statements
*************************************************************
****/
#include<iostream.h>
#include<conio.h>

int main()
{

 int a, b, c, greatest;
 cout<<"Enter the valules for a, b & c \n";
 cin>>a>>b>>c;
```

```
if(a > b)
{
     if(a>c)
     {
               cout<<"\n A is the greatest of three";
     }
     else
     {
               cout<<"\n C is the greatest of three";
     }

}
else
{
     if (b>c)
     {
               cout<<"\n B is the greatest of three";
     }
     else
     {
               cout<<"\n C is the greatest of three";
     }

}
 getch();
 return 0;
}
```

The flowchart for nested if else for the above program is as follows:



6.2.3 Conditional operator:

- It is also known as ternary operator since it takes three parameters.
- It is represented as **? :**
- In one sense it is short form of if-else statement.
- Its syntax is as follows:

```
Expression 1 ?  Expression 2 :   Expression3;
```

- It is read as follows: If Expression 1 is true. i.e non zero, expression 2 is returned else expression 3 is returned.

- Example :

```
int a = 3;
int b = (a > 5 ? 1 ; 0);
```

In the above example is the first line reads an assignment statement where variable a is assigned the value 3.

The second line uses conditional operator. Her if value of a is greater than 5 then b is assigned a value of 1 else a value of zero.

- With the if-else statement the above example could be written as:

```
int a = 3;
int b;
if(a > 5)
    b= 1;
else
    b= 0;
```

- Conditional operators need not be limited to arithmetic expressions only. It can also be used as follows:

```
char gender;
cout<<"enter gender\t";
cin>>gender;
(gender=='m' ? cout<<"Gender is Male" : cout<<"Gender is Female" );
```

If the user enters 'm' then the line printed will be "Gender is Male" else the line printed will be "Gender is Female".

- Nesting of conditional operators is also possible. Ex. Greater of three numbers

```
int a =4, b=5, c=2;
int greater;
greater = ( a > b ? (a > c ? a : c) : ( b > c ? b : c ) );
```

- The only limitation of Conditional operator is that it allows only one statement to executed after ? or :
- Example:

```
/*****************************************************
Program 6.3
Author : Nikhil Pawanikar
Description : Program to take three inputs from user and print the greater
                Of the three using conditional operator
*****************************************************/

#include<iostream.h>
#include<conio.h>
```

```
int main()
{

 int a, b, c, greatest;
 cout<<"Enter the valules for a, b & c \n";
 cin>>a>>b>>c;

 (a > b ? (a > c ? greatest = a  : greatest = c )  : (b > c ? greatest = b  :
greatest = c ));

 cout<<"\nGreatest of three is "<< greatest;
 getch();
 return 0;
}
```

Note: Semicolon is used only once with conditional opertors at the end of statement.

6.2.4 switch statement:

- We use if statements to choose one among the available alternatives.
- When the number of alternatives goes on increasing the complexity to implement also goes on increasing.
- C++ has a multiway decision statement called **switch.**
- The possible values of expression/condition is represented by **case.** A switch statement can have multiple cases representing multiple decisions to be taken.
- The keyword **break** is used inside every case of switch statement to exit the statement once the case is matched and executed.
- The syntax of a switch statement is as follows:

```
switch(integer expression)
{
        case constant1:
                do this;
                break;
        case constant1:
                do this;
                break;
        case constant1:
                do this;
                break;
```

```
            default:
                    do this;
                    break;
    }
```

- The **integer expression** is any expression that will evaulate to give an integer value.
- The keyword **case** is followed by an integer or character constant which may represent the value of integer expression. Each constant character or integer must be unique.
- Every case contains a set of valid c++ statements. The keyword **break** is the last statement inside every case and forces the execution to come out of the switch statement. Without break the execution is said to **fall through** the cases i.e if there is no break the execution proceeds to the next case even if it is not supposed to execute.
- When a program containing a switch statement is executed the integer expression following the switch keyword is evaluated first. This value is then matched one by one with the constant values that follow the case keyword. If a match is found the statements following the case are executed. If no match is found then the statements under the default case are executed.
- Example: Consider the following program that displays a specific message as per the input given by the user.

```
/*************************************************
Program 6.4
Author : Nikhil Pawanikar
Description : This program prompts the user to enter any
              number between 1-5 and displays corresponding
              papers nomenclature.
*************************************************/
#include<iostream.h>
#include<conio.h>

int main()
{
 int number;
 cout<<"Semester 1 has five papers."<<endl;
 cout<<"Enter any number in 1 - 5 to know the nomenclature of
the paper \n";
 cin>>number;
cout<<endl;
 switch(number)
```

```
{
  case 1:
        cout<<"Paper 1 is Professional Communication Skills\n";
        break;
  case 2:
        cout<<"Paper 2 is Applied Mathematics-1 \n";
        break;
  case 3:
        cout<<"Paper 3 is Fundamentals of digital computing  \n";
        break;
  case 4:
        cout<<"Paper 4 is Electronics & Communication
Technology \n";
        break;
  case 5:
         cout<<"Paper 5 is Introduction to C++ Programming -1
\n";
        break;
   default:
         cout<<"Not a valid input";
         break;
 }
 getch();
 return 0;
}
```

Output:

First Run:

```
Semester 1 has five papers.
Enter any number in 1 - 5 to know the nomenclature of the paper
5
Paper 5 is Introduction to C++ Programming -1
```

Second Run:

```
Semester 1 has five papers.
Enter any number in 1 - 5 to know the nomenclature of the paper
8
Not a valid input
```

- In the program above the user is supposed to enter any number between 1 – 5 which stands for the papers at FYBScIT Sem I. The value entered by the user is stored in a variable number which

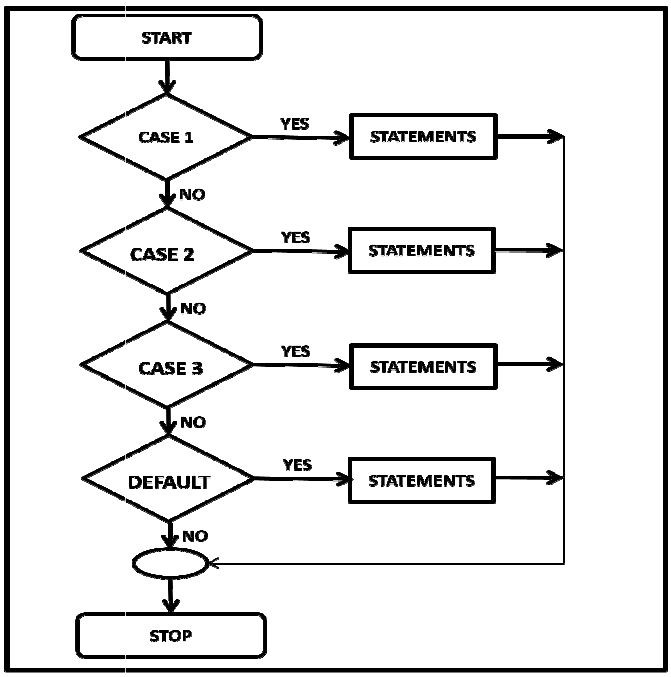
also happens to be the integer expression following the keyword switch.

- Since we have 5 subjects at sem I of FYBScIT we have written 5 cases. Here integer constants are used since the integer expression is an integer. If the expression was character then the case constants would also have to be characters.
- In the first run when the user enters 5, 5 is stored in the variable number. Then the value of variable number is matched with every case constant inside switch statement in the order in which they appear. Since a match is found that equals to 5, the statements following that case are executed.
- When the user enters 8 as an input there is no match with the case constants so the default case is executed.
- The same program with character input and character constant would look like this:

```
/*********************************************************
Program 6.5
Author : Nikhil Pawanikar
Description : This program prompts the user to enter any
              character between A-E and displays corresponding
              papers nomenclature.
*********************************************************/

#include<iostream.h>
#include<conio.h>
int main()
{
 char choice;
  cout<<"Semester 1 has five papers."<<endl;
 cout<<"Enter any character from A - E to know the nomenclature of the
paper \n";
 cin>>choice;

 switch(choice)
 {
    case 'A':
         cout<<"Paper A is Professional Communication Skills\n";
         break;
```

```
  case 'B':
        cout<<"Paper B is Applied Mathematics-1 \n";
        break;

case 'C':
        cout<<"Paper C is Fundamentals of digital computing  \n";
        break;

case 'D':
        cout<<"Paper D is Electronics & Communication Technology \n";
        break;

case 'E':
        cout<<"Paper E is Introduction to C++ Programming -1 \n";
        break;

default:
        cout<<"Not a valid input";
        break;
}
getch();
return 0;
}
```

The flowchart for switch statement is as follows:



6.3 COMPOUND STATEMENTS

- The statements that are written inside a pair of parenthesis { }, are called compound statements.
- We have already seen the use of compound statements in if, if-else and switch statements.
- Ex.

```
int a,b,c;
a=2,b=3,c=0;
if(a!=0)
{
c=a+b;
cout<<"c="<<c;
}
```

Here the statements written inside parenthesis { } are compound statements, once the control passes into this block all the statements inside it are executed unless forced to quit or jump out before reaching the last statement.

Compound Conditions:

- Just as we have compound statements we have compound conditions too.

- Compound conditions are used to combine two or more conditions using logical operators && (and), || (or) and ! (not).
- They are defined as:

A && B - Evaluates to true if both A and B are true

A || B - Evaluates to true if either A or B is true or in other words
Evaluates to false if and only if both A & B are both false

! A - Evaluates to true if and only if A evaluates to false

- Using truth tables it could be defined as follows:

A	B	A&&B	A||B	!A
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

Example : Program to find the greater of 3 numbers

```
/*******************************************************
Program 6.3
Author : Nikhil Pawanikar
Description : Program to take three inputs from user and print the greater
                Of the three using nested if-else statements
*******************************************************/
#include<iostream.h>
#include<conio.h>

int main()
{
     int a, b, c, greatest;
     cout<<"Enter the valules for a, b & c \n";
     cin>>a>>b>>c;

    if(a > b && a >c)
    {
            cout<<"\n A is the greatest";
    }
    if (b>a  && b > c)
```

```
    {
            cout<<"\n B is the greatest of three";
    }
    if (c>a  && c > b)
    {
            cout<<"\n C is the greatest of three";
    }
     getch();
     return 0;
}
```

6.4 INCREMENT (++) & DECREMENT(--) OPERATORS

- These are unary operators i.e. they take only one operand.
- Increment & Decrement operators are used to increase or decrease the value of integer variables or integer constants by 1.
- There are two forms of Increment operator: Pre-increment & Post-increment.
- Example: an integer variable m can be incremented in two ways:
  m++; ++m;
  & decremented in two ways m--; --m;

```
/*******************************************************
Program 6.
Author : Nikhil Pawanikar
Description : Program to display the use of Increment & Decrement
operators
*******************************************************/
#include<iostream.h>
#include<conio.h>
int main()
{
 int a, b;
 a=0;
 b=2;
 clrscr();
 a = ++b;
 cout<<" value of a is "<<a;          //value of a is 3
 cout<<" value of b is "<<b;          //value of b is 3
 b=4;
 a=b++;
 cout<<" value of a is "<<a;          //value of a is 4
```

```
 cout<<" value of b is "<<b;          //value of a is 5
 a = --b;
 cout<<" value of a is "<<a;          //value of a is 3
 cout<<" value of b is "<<b;          //value of b is 3
 b=4;
 a=b--;
 cout<<" value of a is "<<a;          //value of a is 4
 cout<<" value of b is "<<b;          //value of a is 3
 getch();
 return 0;
}
```

6.5 Review Questions

1. Explain the different forms of if statement.
2. Explain the difference between if statement & switch statement.
3. What happens if we do not use break in switch statement?
4. Write a program in C++ for the following:
    1. Take an integer number from the user and find if it is even.
    2. Take an integer number from the user and find if it is odd or even.
    3. Take the value of cost price and selling price of a particular item from user and print if it profit or loss and how much.
    4. Take the value of 5 subjects from user. Calculate sum, percentage and display grade(use compound conditions)
    5. Using switch case, write a program to perform arithmetic operations (+,-,* ,/). Show a menu to the user and allow selecting an option.

6.6 Reference & Further Reading

1. Let us C – Yashwant Kanetkar, Chapters 2 & 4.
2. Object Oriented Programming with C++ - E. Balaguruswamy, Chapter 3
3. Programming in C++, Schaums Outlines – John R. Hubbard, Chapters 2 &3.

Chapter 7

Looping Flow of Control

Unit Structure

7.1 INTRODUCTION

In the previous chapter we had seen the concept of Branching Flow of control which involves making decision which involves having a condition. If the condition is satisfied then the decision is to execute a particular set of instructions, if not then another set of instructions.

This chapter is dedicated to Looping or iteration. It is used when a set of instructions is to be executed multiple times or for a fixed number of times until a particular condition is satisfied.

This repetitive action is done through Loop Control Instruction. C++ offers the following Loop Control Instructions:

1. for
2. while
3. do while

7.2 LOOP CONTROL INSTRUCTIONS

7.2.1 while loop:

- The while loop has the following syntax:

while (condition is true)
statement;
	Initialize counter variable;
	while (condition is true)
	{
	statement;
	statement;
	increment counter;

	}

- Here condition is an integer expression just like the one we have seen in the previous chapter.
- When the system reads the keyword while it evaluates the expression that follows.

  If the expression is evaluated as true the statement following the condition is executed. It may be a simple statement or a compound statement enclosed in parenthesis { }.
- Example : consider the following example to find the cube of the number entered by user.

```
/*******************************************************
Program 7.1
Author : Nikhil Pawanikar
Description : Program to display sum of numbers upto n
*******************************************************/
#include<iostream.h>
#include<conio.h>
int main()
{
 int num,sum=0;;
 clrscr();

 cout<<" \n Enter any positive integer number \n ";
 cin>>num;
int i=0;

 while(i<=num)
        sum=sum + i++;
cout<<"\n Sum of numbers till  "<<num<<" = "<<sum;

 getch();
 return 0;
}
```

Output

First Run

```
Enter any positive integer number
2
Sum of numbers till  2 = 3
```

Second Run

```
Enter any positive integer number
5
Sum of numbers till  5 = 15
```

- In the above example the while loop will execute a single statement until the condition i<=num evaluates to true.

- The following example executes a set of statements once the condition becomes true.

```
/*************************************************************
**************
Program 7.2
Author : Nikhil Pawanikar
Description : Program to display the cube of a number until the user
types zero
*************************************************************
*************/
#include<iostream.h>
#include<conio.h>

int main()
{

 int num,cube=0;;
 clrscr();


 int i=0;
 while(i<=num)
 {
        cout<<" \n Enter any positive integer number to find its cube,
type 0 to exit \n ";
        cin>>num;
        cube = num * num * num;
        cout<<"\n Cube of num "<<num<<" = "<<cube;
        i=i+1;
 }

 getch();
 return 0;
}
```

Output

First Run

```
Enter any positive integer number to find its cube, type
0 to exit
2
Cube of 2 = 8
```

Second Run

```
Enter any positive integer number to find its cube, type
0 to exit
```

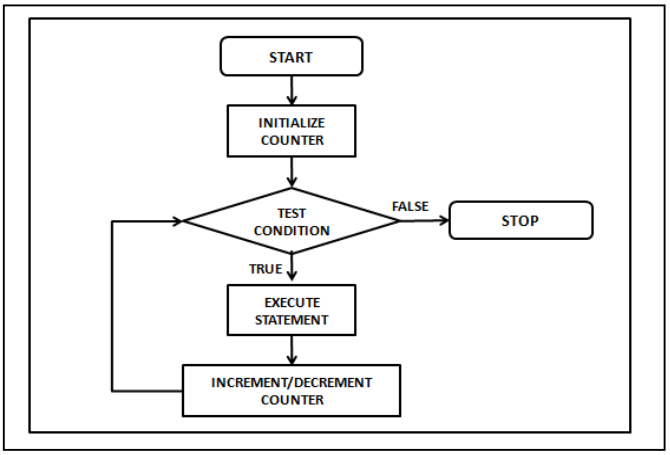
```
0
Cube of 0 = 0
```

Note: The while loop may execute infinite number of times if the condition in the loop happens to be true for every value.

Ex.

```
while (1)
{
cout<<"\nhello world";
}
```

- Since the condition following while returns a non zero value it happens to be true every time a condition is checked and runs infinitely.
- Instead of incrementing the counter value we can also decrement it.
- The flowchart for a while loop is as follows:



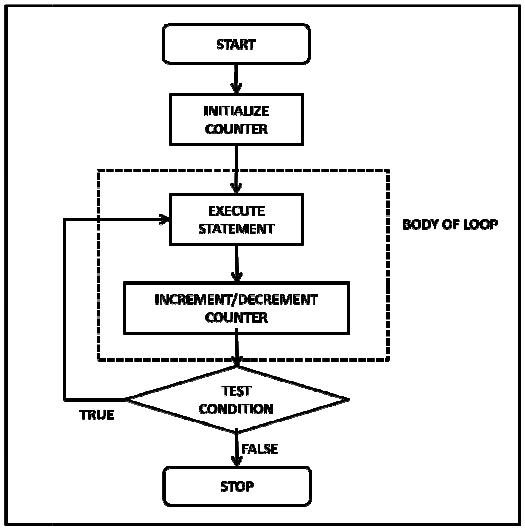
7.2.2 do while

- The syntax for the do-while loop is as follows:

```
do
   statement;
while (condition is true);
```

```
   Initialize counter variable;
do{
      statement;
      statement;
      increment counter;
   } while (condition is true);
```

- The do while loop executes in the same way as the while loop. The only difference is that in do-while, the statements are executed at least once even if the condition is evaluated to be false.
- The do while statement is always terminated with a semicolon (;) after the loop ends.
- The following flow chart will help understand the do-while loop:



- As shown in the flowchart above, the body of the loop is executed and then the condition is tested. So when the program executes the first iteration will always execute the loop and then test for the condition. If the condition evaluates to true it will again execute the block of statements else it will exit the loop and execute the next statement following the do-while loop if any.
- **Example:** Consider the program discussed for while loop. Cube of number. We shall see how it could be done using do-while loop.

```
/*******************************************************
Program 7.3
Author : Nikhil Pawanikar
Description : Program to display the cube of a number until the user
wants to using
Do-while loop
*******************************************************/
#include<iostream.h>
#include<conio.h>

int main()
```

```
{
 int num,cube=0;;
 char ans;
 clrscr();
  int i=0;
 do
 {
   cout<<" \n Enter any positive integer number to find its cube \n ";
   cin>>num;
     cube = num * num * num;
     cout<<"\n Cube of num "<<num<<" = "<<cube;
     i=i+1;
     cout<<"\nPress Y to continue";
     cin>>ans;
}while(ans=='y' || ans == 'Y');

  return 0;
}
```

Output

First Run

```
Enter any positive integer number to find its cube
2
Cube of  2 = 8
Press Y to continue
Y
Enter any positive integer number to find its cube
4
Cube of  4 = 64
Press Y to continue
a
```

Second Run

```
Enter any positive integer number to find its cube
3
Cube of  3 = 27
Press Y to continue
r
```

- In the above example the condition is on whether the user wants to continue the program or exit.
- In first run, the program executes the loop for one time and then checks the condition where the user enters ‘y’ and executes again. Next time when the user was prompted to continue, a key other than y was pressed and hence the condition was false and the control came out of the loop.
- In the second run, the program executes the loop for one time and then checks the condition where the user enters ‘y’ and executes again. . Next time when the user was prompted to

continue, a key other than y was pressed and hence the condition was false and the control came out of the loop.

7.2.3 for

- The syntax of the for loop is as follows:

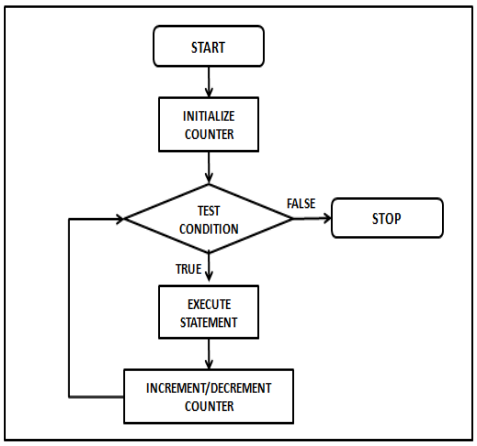
```
for (initialization ; test condition ; increment/decrement loop counter)
        statement;
```

or

```
for (initialization ;  condition ; increment/decrement loop counter)
 {
    statement;
    statement;
 }
```

- The for loop allows to specify three things in a single line:
  1. Create and initialize loop counter variable
  2. Test condition
  3. Increase or decrease the value of loop counter after every iteration

- The flowchart of for loop is same as while loop.



- Example:The program for calculating sum of numbers till n using for loop is as follows:

```
/*************************************************
Program 7.4
Author : Nikhil Pawanikar
Description : Program to display sum of numbers upto n
*************************************************/
#include<iostream.h>
#include<conio.h>
int main()
{
    int num,sum=0;;
   clrscr();

   cout<<" \n Enter any positive integer number \n ";
   cin>>num;
   for(int i =0; i<=num ; i ++)
{
   sum=sum + i;
}

cout<<"\n Sum of numbers till  "<<num<<" = "<<sum;
getch();
return 0;
}
```

Output:

First Run

```
Enter any positive integer number
2
Sum of numbers till  2 = 3
```

Second Run

```
Enter any positive integer number
```

```
5
Sum of numbers till  5 = 15
```

- In the program above, for statement first initializes the variable i to zero, then it checks for the condition. If the condition is true the statements inside the block following for statement are executed. Once the loop is over the value of i is incremented by one and the condition is tested again.
- As long as the value of i happens to be less than or equal to num the loop will execute. When the value of i becomes greater than num the control is passed to the next statement after the loop.

Nesting of Loops:

Nesting of loop control structures is possible in the same way as in decision control.

7.3 REVIEW QUESTIONS

1. List and explain the statements under Loop Control Structure.
2. Explain the difference between the while and do-while loop.
3. Explain the difference between the while and for loop.
4. Write a program in C++ to do the following:
5. Take a number from user and print Fibonacci series on the screen containing that many numbers(i.e. 0 ,1, 1, 2, 3, 5, 8, 13, 21)
6. Print the odd and even number from 1 to 100
7. Print the following pattern

```
*                1
* *              1 2
* * *            1 2 3
* * * *          1 2 3 4
* * * * *        1 2 3 4 5
```

7.4 REFERENCES & FURTHER READING

1. Let us C – Yashwant Kanetkar, Chapter 3.

2. Object Oriented Programming with C++ - E. Balaguruswamy, Chapter 3
3. Programming in C++, Schaums Outlines – John R. Hubbard, Chapter 4.
4. Theory & Problems of Programming with c ++ – John R. Hubbard, Chapter 3.

Chapter 8

Break and continue

Unit Structure

8.1 Introduction

8.2 Break and continue.

8.3 Manipulators: endl , setw,sizeof.

8.4 Type Cast Operators

8.5 Scope resolution operators

8.1. INTRODUCTION

In the previous chapter we studied Loop control structure. In this chapter we study two mechanisms to bypass the looping construct.

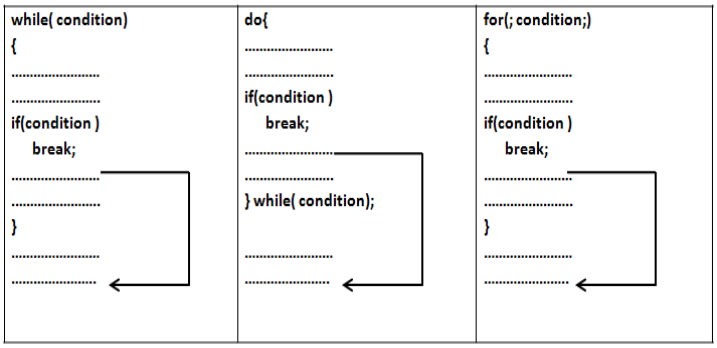
These are break and continue.
Then we study the use of following manipulators:
endl , setw,sizeof & finally operators like
Type Cast Operators & Scope resolution operators

8.2 BREAK AND CONTINUE

break

- Normally a **while** loop, a **do..while** loop, or a **for** loop will terminate only at the beginning or at the end of the complete sequence of statements in the loop's block.
- But sometimes there may be situations when you require the control to exit the loop. This can be fulfilled by using the **break** keyword.
- **A break statement terminates a loop**. The control is transferred to the first statement immediately following the loop construct.
- We have seen the use of break statement in switch statement. When a match is found the statements under the case are executed and then break statement causes the switch loop to terminate.
- A break statement is usually associated with a condition, so there will be an if statement.

The general form of break is as follows:



- Example: Consider program 7.1. to print sum of integers upto n to be done using break

```
/*************************************************
*****
Program 8.1
Author : Nikhil Pawanikar
Description : Program to display sum of numbers upto n using
break
*************************************************
****/
#include<iostream.h>
#include<conio.h>
int main()
{
 int num,sum=0;;
 clrscr();
cout<<" \n Enter any positive integer number \n ";
cin>>num;
int i=0;
while(1)
{
if(i>num)
            break;
sum=sum + i++;
}
cout<<"\n Sum of numbers till  "<<num<<" = "<<sum;
 getch();
 return 0;
}
```

Output

First Run

```
Enter any positive integer number
```

```
2
Sum of numbers till  2 = 3
```

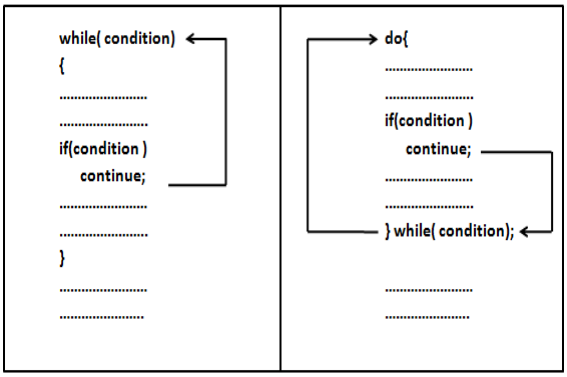
Second Run

```
Enter any positive integer number
5
Sum of numbers till  5 = 15
```

- The above program is same as program 7.1. Here the body of while loop runs infinitely. The only condition implemented is (i>num) if this condition evaluates to true the control will exit the while loop and proceed to execute the statements after the loop.

Continue:

- Sometimes there may be situations when you require the control to exit the loop or skip the particular iteration and go to the next. This can be done using the keyword **continue.**
- A **continue** statement causes the rest of the body of the loop to be omitted, for the current iteration. The control is transferred to the code that evaluates the normal test condition for loop termination.
- The continue statement is similar to the break statement but instead of terminating the loop, it transfers execution to the next iteration of the loop. It continues the loop after skipping the remaining statements in its current iteration.
- The general form of continue is as follows:



Example: Program to print the following pattern

1

1 2

1 2 3 4

1 2 3 4 5

Here the third line in the sequence is missing: 1 2 3

This pattern can be printed on the screen by using continue

```
/*****************************************
Program 8.2
Author : Nikhil Pawanikar
Description : Program to display the following pattern
using continue
                1
                1 2
                1 2 3 4
                1 2 3 4 5
*****************************************/
#include<iostream.h>
#include<conio.h>
int main()
{
 //int num,sum=0;;
 clrscr();

for(int i=0; i<=5 ; i++)
{
  if (i==3)continue;
  for(int j=1;j<=i;j++)
  {
            cout<<j;
  }
  cout<<endl;
}

 getch();
 return 0;
}
```

8.3 MANIPULATORS: ENDL , SETW ,SIZEOF.

The operators that are used to format the output that is supposed to be displayed on the screen are called Manipulators.

1. endl manipulator:

It is used to skip the current line and go to the next line. It is same as \n

2. setw() manipulator:

It is used to right align the output.

To use setw() manipulator, the header file <iomanip> has to be included in the program.

Example: Consider the program to input the contact information of a person and display the output formatted to the right.

```
/*******************************************************
Program 8.3
Author: Nikhil Pawanikar
Description: Program to display the use of manipulators setw() and
endl
*******************************************************/
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
#include<string.h>
int main()
{
 char name[10],gender[6],location[10],contact_no[10];
 clrscr();
 cout<<"enter name:\t";
 cin>>name;
 cout<<"\nenter gender (m/f):\t";
 cin>>gender;
 cout<<"\nEnter Location: \t";
 cin>>location;
 cout<<"\nContact Number:\t";
 cin>>contact_no;
 cout<<"\NAME                    :"<<setw(15)<<name<<endl;
 cout<<"\nGENDER                  :"<<setw(15)<<gender<<endl;
 cout<<"\nLOCATION                :"<<setw(15)<<location<<endl;
 cout<<"\NCONTACT NUMBER          :"<<setw(15)<<contact_no;
 getch();
 return 0;
}
```

Output

```
enter name:       NIKHIL P
nenter gender (m/f):      MALE
Enter Location:    SANTACRUZ
Contact Number:       9988776655
```

NAME							N	I	K	H	I	L		P
GENDER											M	A	L	E
LOCATION							S	A	N	T	A	C	R	U	Z
CONTACT NUMBER						9	9	8	8	7	7	6	6	5	5

sizeof() operator

The **sizeof** operator yields the size of its operand with respect to the size of type **char**

Example:

```
/*******************************************************
Program 8.3
Author: Nikhil Pawanikar
Description: Program to display the use of manipulators setw() and
endl
*******************************************************/
#include <iostream.h>
#include<conio.h>
int main()
{
  char var_name[] = "INSTITUE OF DISTANCE & OPEN LEARNING!";

  cout  << "The size of a char is: "
          << "\nThe length of " << var_name << " is: "
          << sizeof( var_name);
  getch();
 return 0;
}
```

Output

```
The length of INSTITUE OF DISTANCE & OPEN LEARNING! is
38
```

8.4 TYPE CAST OPERATORS

- C++ provides Type cast **operators** to explicitly convert the type of variables or expression from one type to another.
- The general syntax is

```
type-name(expression)
```

- Here type-name is the target datatype we want to convert the variable expression into.
- Example: float p **= float(a);**
- **A typename behaves as if it is a function for converting a variable type into desired target type.**
- C++ has introduced the following new type cast operators:
  1. const_cast
  2. static_cast

3. dynamic_cast
4. reinterpret_cast

8.5 REVIEW QUESTIONS

1. Explain the use of break and continue with examples.
2. Explain the difference between break & continue.
3. Explain the use of Manipulators endl & setw.
4. Write short note on sizeof operator.

8.6 REFERENCES & FURTHER READING

1. Let us C – Yashwant Kanetkar, Chapter 3.
2. Object Oriented Programming with C++ - E. Balaguruswamy, Chapter 3
3. Programming in C++, Schaums Outlines – John R. Hubbard, Chapter 2.

9

Chapter 9
Introduction to Functions

Unit Structure

9.1. INTRODUCTION

Usually programs are much larger than the programs that we have seen so far. To make large programs manageable and less complicated, they are broken down into subprograms. These subprograms are called functions.

The basic principle of Functions is ***DIVIDE AND CONQUER***. Using functions we can divide a larger task into smaller subtasks that are manageable.

9.1.1. What is a function?

1. A function is a self contained block of statements.
   A function is self contained in the sense it may have its own variables and constants

2. It is designed to do a well defined task.

Since a function is a part of a larger program (i.e. a subprogram) it has a particular job to perform.

3. It has a name.
   A function can be used (invoked) by the name given to it.

4. It may have return type
   A function invoked by a calling program may or may not return a value to it. In case it returns a value the functions return type is the same as the variables data type.

5. A program that has functions should have the following three things in it:
   a. Function Declaration or Prototype
   b. Function Call
   c. Function definition, which are discussed in a later part of this chapter.

9.1.2. Why use it?

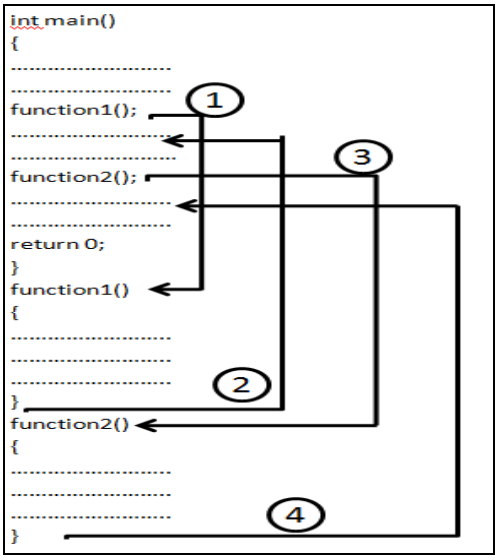
1. Functions are a structured way to programming. Larger programs get divided into smaller manageable units.

2. If a specific block of statements has to be executed multiple times (for example. taking contact details from 100 users), it can be written as a function and that function can be repeatedly executed. This implies that redundancy in writing the same piece of code multiple times is removed.

3. Dividing a large program into smaller subprograms using functions help to easily code them and reduces the debugging effort.

9.1.3. How does it work in an program?

Consider the following:



The program to the left contains three functions. First one is the main() function, second is function1() and third is function2().

The execution of any program begins with the execution of main function. Unless there is a decision or looping construct the execution of the program proceeds in a serial manner.

In the diagram to the left, the program execution begins with main(), all the statements get executed.

A function gets called when the function name is followed by a semicolon.

A function is defined when function name is followed by a pair of braces in which one or more statements may be present.

When the system encounters a call to function1() the program control jumps outside the main() function to execute the block of statements named function1() shown by arrow number1.

Once the last statement in function1() is executed the program control is again transferred to main() and the immediate statement after main is executed, shown by arrow number2.

When the system encounters a call to function2() the program control jumps outside the main() function again to execute the block of statements named function2() shown by arrow number 3.

Once the last statement in function2() is executed the program control is again transferred to main() and the immediate statement after main is executed, shown by arrow number 4.

9.2. TYPES OF FUNCTIONS

Functions are of two types

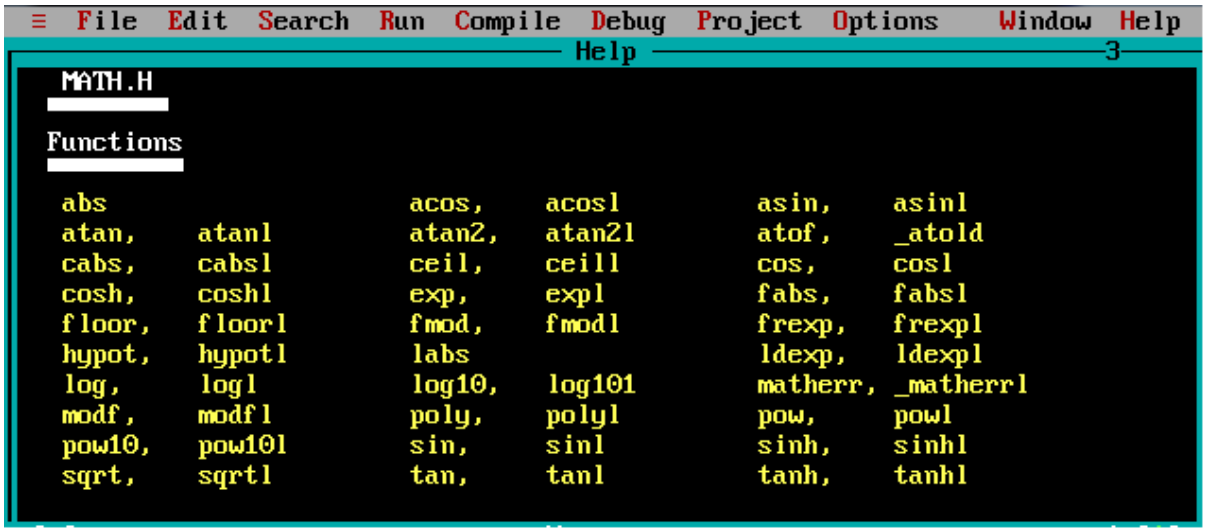
1. Built-in functions
2. User defined functions

9.2.1 Built-in Functions:

- These are also called Standard Library Functions. As the name suggests it is a Library of functions. These are the functions that are already present .i.e. predefined in the system.
- They have been written, compiled and placed in libraries under header files.
- We can use these functions in our programs by just specifying the name of the **header files** that contains the function of our interest.
- Example: we have already used a built-in function in chapter 8, the setw() manipulator. To use this function we have to add the header file <iomanip.h> to our program. The function definition for setw() is compiled and placed in the header file <iomanip.h>. If we do not include this file in our program and still use setw(), the compiler will return a **FUNCTION PROTOTYPE MISSING** error.
  For code refer to example on Page **>>pls enter page no of program 8.3 chapter 8<<**

9.2.1.1. Math Library Functions

- C++ provides a library of math related functions called **Math Library Functions.**
- These functions are placed in header file <math.h> and it contains 59 functions.
- The following is a snapshot of the help menu of Turbo C++ displaying the list of available built-in functions under <math.h>.



Example : Print Square Root of Numbers from 1 to 10

```
/*************************************************************
**************
Program 9.1
Author: Nikhil Pawanikar
Description: Program to display the use of math library functions
*************************************************************
**************/
#include <iostream.h>
#include<conio.h>
#include<math.h>c

int main()
{
  cout<<"Number"<<"\tSquare Root\n";
  for(int i=1;i<=10;i++)
        {
                cout<<i<<"\t"<<sqrt(i)<<endl;
        }
  getch();
 return 0;
 }
```

Output

Number	Square Root
1	1
2	1.414214
3	1.7320513
4	2
5	2.236068

6	2.44949
7	2.645751
8	2.282427
9	3
10	3.162278

- This program prints the square roots of the numbers 1 through 10. The value of variable i from the loop counter is passed to sqrt(i).
- i is called the parameter passed to function sqrt.
- Each time the expression sqrt(x)is evaluated in the for loop, the sqrt() function is executed for the value of i passed to it.
- Its actual code is hidden away within the Standard C++ Library.
- Following are some of the functions available under the header file <math.h> and their uses:

Trigonometric functions:	
cos	Compute cosine (function)
sin	Compute sine (function)
tan	Compute tangent (function)
acos	Compute arc cosine (function)
asin	Compute arc sine (function)
atan	Compute arc tangent (function)
atan2	Compute arc tangent with two parameters (function)
Hyperbolic functions:	
cosh	Compute hyperbolic cosine (function)
sinh	Compute hyperbolic sine (function)
tanh	Compute hyperbolic tangent (function)
Power functions	
pow	Raise to power (function)
sqrt	Compute square root (function)
Rounding, absolute value and remainder functions:	
ceil	Round up value (function)
fabs	Compute absolute value (function)
floor	Round down value (function)
fmod	Compute remainder of division (function)
Exponential and logarithmic functions:	
exp	Compute exponential function (function)
frexp	Get significand and exponent (function)
ldexp	Generate number from significand and exponent (function)
log	Compute natural logarithm (function)
log10	Compute common logarithm (function)
modf	Break into fractional and integral parts (function)

9.2.2. User defined functions:

These are the functions other than the Standard Library Functions. These are created by the users and the user has the flexibility to choose the function name, the statements that will be executed, the parameters that will be passed to the user & the return type of the function.

Any program using functions will have the following three necessary things:

1. **Function prototype or function declaration**
   It is the name of the function along with its return-type and parameter list.

2. **Function call**
   Any function name inside the main() followed by semicolon (;) is a Function Call.

3. **Function Definition:**
   A function name followed by a pair of parenthesis {} including one or more statements.

- In case of built-in functions, function prototype and function definition are not necessary, they have been already declared and defined in the libraries.
- Consider the following example:

```
return-type function1();

int main()
{
……..
……..
function1();

……..

……..

return 0;

}
```

```
function1()

{

……..

……..

return();

}
```

- The first line of the above example **return-type function1();** is called the **function prototype or function declaration**. It is used to declare the function to the compiler.
  This statement is always written outside(before) the main().

- The statement **function1()** along with the statements in the parenthesis shown below is called the **function definition.** The function definition contains the instructions to be executed when the function is called.

```
function1()
{
……..
…….
}
```

Function definition is always done outside the main().

- The statement **function1();** inside main() is a function call.
  A function gets called when the function name is followed by a semicolon.
  When this statement function1(); is executed the program control gets transferred to the the function definition of function1() which is outside the main(). All the statements inside function1() are executed and then the control gets transferred to the next statement after the function call.

Note:
From this point onwards Function prototype & Function definition means prototype & definition for a user-defined function, Since only user-defined functions have function prototype/ declaration and function definition.

Function Definition:

- A function is defined when function name is followed by a pair of braces in which one or more statements may be present.

- A function definition has 2 parts
  1. Function head
  2. Function Body

- Example :

```
int square(int x)
{
return x*x;          //          returns
square of x:
}
```

- The function returns the square of the integer passed to it. Thus the call square(3) would return 9.

1. Function Head

- The syntax for the **head** of a function is

```
return-type
name(parameter-list)
```

- The above statement tells the compiler three things about the function:
  i. **Name** of the function
  ii. Its **return-type** i.e type of value to be returned by the function
  iii. Its **parameter list**.

  In the example shown above the head of the function is:

```
int square(int x)
```

  i. **square** is the name of the function,
  ii. **int** is the type of value that the function is returning to main()
  iii. and the parameter list **(int x)** contains a single parameter that is passed to the function square by the main()

2. Function Body

- The **body** of a function is the block of code that follows its head.
- It contains the code that performs the function's action.
- It includes the **return** statement that specifies the value that the function sends back to the place where it was called usually main().
- The body of the square function is

```
{
return x*x;          //          returns
square of x:
}
```

9.2.2.1. Local Variables in Functions

A variable can be declared inside a function definition, but it would be only local to the function. It cannot be used anywhere outside the function.

They exist only when the function is executing. The variables in the parameter list of function definition are called formal arguments and they are also local variables and exist only for the duration of the function execution.

9.2.2.2. Function Prototypes:

- The general syntax of a function prototype is

```
return-type     function-name
(parameter list);
```

- The above statement tells the compiler three things about the function:
  1. **Return-type** i.e type of value to be returned by the function
  2. **Name** of the function
  3. **Parameter list**. (the number of parameters the function will receive and their data-types).

- A Function Prototype is terminated by a semicolon

- Example: The complete program for finding square of a program is written as follows:

```
/*************************************************************
**************
Program 9.2
Author: Nikhil Pawanikar
Description: Program to display the square of a number entered by user.
             (Demonstrate the concept of function prototype)
*************************************************************
**************/
#include <iostream.h>
#include<conio.h>
int square(int m);                                  // Function Prototype
int main()
{
  int num, sqr=0;
```

```
  cout<<"\nEnter number to find its Square"<<"\t ";
  cin>>num;
  sqr=square(num);                                    // Function call
  cout<<"\nSquare of "<<num<<" = " <<sqr;
  getch();
  return 0;
 }

int square(int x)                                     // Function definition
{
return x*x;              // returns square of x:
}
```

Output:

First Run

```
Enter number to find its Square        5
Square of 5 = 25
```

Second Run

```
Enter number to find its Square
Square of 8 = 64
```

- The statement below is called **function declaration or function prototype.**

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

- The function prototype in the program above also contains the same:
    1. The function would return an **integer** value, hence, its return type is **int**
    2. The name of the function is **square**
    3. The function receives one parameter of type integer from the place where it is called from i.e. main().

- Following are some examples of function declaration:

```
float area(float length, float
breadth);
```

```
float perimeter(float side1,
float side2);
```

- Inside the function declaration each variable must be declared independently, the following declaration is invalid

```
float sum_of_angle(int angle1, int
angle2, angle3);
```

- The parameter names are optional in a function declaration, they are simply **dummy** variables.

```
float sum_of_angle(int, int, int);
```

- The above is valid since a Function prototype expects only the number of parameters and its data-types from the parameter list.

  For every function to be used there should be a function prototype. During program execution when the compiler encounters a function call, it first matches the prototype having the same number and type of arguments and then calls the appropriate function for execution.
- A function prototype is different from a function call, it(function call) does not indicate the return-type of the function.
- **Actual & Formal arguments:** In the program above, the statement sqr= square(num);

  The variable num being passed to the function square is called actual parameter & and the variable x in the function head of function square is called formal parameter.

9.3. FUNCTION OVERLOADING

- Overloading means using one thing for different purposes. C++ supports function overloading.
- Function overloading is also called **FUNCTION POLYMORPHISM**. Under this, the same function name can be used to create multiple function definitions to perform different tasks.
- It means that we can have a set of functions that have the same name but different signatures. A function signature includes its return type and parameter list.

- Example: an overloaded function multiply() is shown below with possible function prototypes and associated function calls and function definitions.

```
//function declarations
        int multiply(int m, int n);                     //prototype 1
        int multiply(int m, int n, int p);              //prototype 2
        double multiply(double m , double n);           //prototype 3
        double multiply(double m , int n);              //prototype 4
        double multiply(int m, double n);               //prototype 5
```

```
//function calls
        int mul = multiply(10, 20);
        int mul = multiply(10, 20,3);
        double mul = multiply(2.5, 3.5);
        double mul = multiply(1.2, 3);
        double mul = multiply(2, 3.5);
```

```
//function definitions

int multiply(int m, int n)
{
        return (m*n);
}

int multiply(int m, int n, int p)
{
        return (m*n*p);
}

double multiply(double m , double n)
{
        return (m*n);
}

double multiply(double m , int n)
{
        return (m*n);
}
double multiply(int m, double n)
{
        return (m*n);
}
```

When a function call is encountered the compiler tries to select the best function to be executed. To do this the compiler matches the prototype having the same number and type of arguments as in the function call and then invokes the appropriate function for execution.

9.4 REVIEW QUESTIONS

1. What is a function?
2. Explain types of functions?
3. Explain function prototyping.
4. Explain user-defined functions
5. Explain built-in functions with examples

9.5 REFERENCES & FURTHER READING

1. Let us C – Yashwant Kanetkar, Chapter 5.
2. Object Oriented Programming with C++ - E. Balaguruswamy, Chapter 4
3. Programming in C++, Schaums Outlines – John R. Hubbard, Chapter 5.

Solved Example:

Program to do arithmetic operations using function and switch case.

```
#include <iostream.h>
#include <conio.h>
#include <process.h>

int getinput();
void arith(int);
void getdata();
void add();
void sub();
void mul();
void div();

double a, b;
int main()
{
        clrscr();
        int choice;
```

```
        while(1)
        {
        clrscr();
        choice= getinput();
        arith(choice);
        getch();
        }
        return  0;
}
int getinput()
{
        int input;
        cout<<"\n Select the operation you want to perform\n"
           <<"1. Addition\n"
           <<"2. Subtraction\n"
           <<"3. Multiplication\n"
           <<"4. Division\n"
           <<"5. Exit\n";
        cin>>input;
        return input;
}

void arith(int choice)
{
                switch(choice)
                {
                        case 1:
                                add();
                                break;
                        case 2:
                                sub();
                                break;
                        case 3:
                                mul();
                                break;
                        case 4:
                                div();
                                break;
                        case 5:
                                exit(1);
                        default:
                                cout<<"\n Please select a proper
input";
                                break;
                }
```

```
}
void getdata(double &a, double &b)
{
        cout<<"\nEnter the values of a and b\t";
        cin>>a>>b;
}

void add()
{
        getdata(a,b);
        cout<<"\n Result of Addition is  : "<<a+b;

}
void sub()
{
        getdata(a,b);
        cout<<"\n Result of Subtraction is  : "<<a-b;
}
void mul()
{
        getdata(a,b);
        cout<<"\n Result of Multiplication is  : "<<a*b;
}
void div()
{
        getdata(a,b);
        cout<<"\n Result of division is  : "<<a/b;
}
```

10

Chapter 10

Unit Structure

10.0 OBJECTIVES

10.1. INTRODUCTION

Now that we know what is a function? And how to use it? We can proceed to some advanced topics related to functions. There are two ways in which a function can be called, they are:

1. Call by value
2. Call by reference.

10.2. CALL BY VALUE

- The examples that we have seen above are all examples of call by value. In this method of calling a function we pass the value of variables to the function as parameters.
- Such function calls are called ‘call by value’. In call by value the changes made to the formal parameters do not change the actual parameters.
- The called function creates a new set of variables and copies the values of actual arguments into formal arguments.
- The function does not have access to the variables declared in the calling program and can only work on the copies of values i.e. the formal arguments.
- Example: Consider the following program for swapping of two numbers.

```
/*************************************************************
**************
Program
Author: Nikhil Pawanikar
Description: Program to swap the values of two numbers.
*************************************************************
**************/
#include <iostream.h>
#include <conio.h>
void swap(int,int);                    //prototype
int main(void)
{
        int a,b;
        cout << "Please enter 2 positive integers:\t ";
        cin >> a>>b;

        cout<<"\n Values before swapping are (in main ()):\n a = "
           <<a<<"\t b = "<<b<<endl;

        swap(a,b);                     //call by value, actual arguments

        cout<<"\n Values after swapping are (in main ()):\n a = "
           <<a<<"\t b = "<<b<<endl;
        getch();
        return 0;
}
void swap(int m,int n)                  //definition, formal arguments
{
        int temp;

        cout<<"\n Values before swapping are (in swap ()):\n m = "
           <<m<<"\t n = "<<n<<endl;
        temp = m;
        m = n;
        n = temp;

        cout<<"\n Values after swapping are(in swap ()):\n m = "
           <<m<<"\t n = "<<n<<endl;
}
```

Output:

First Run

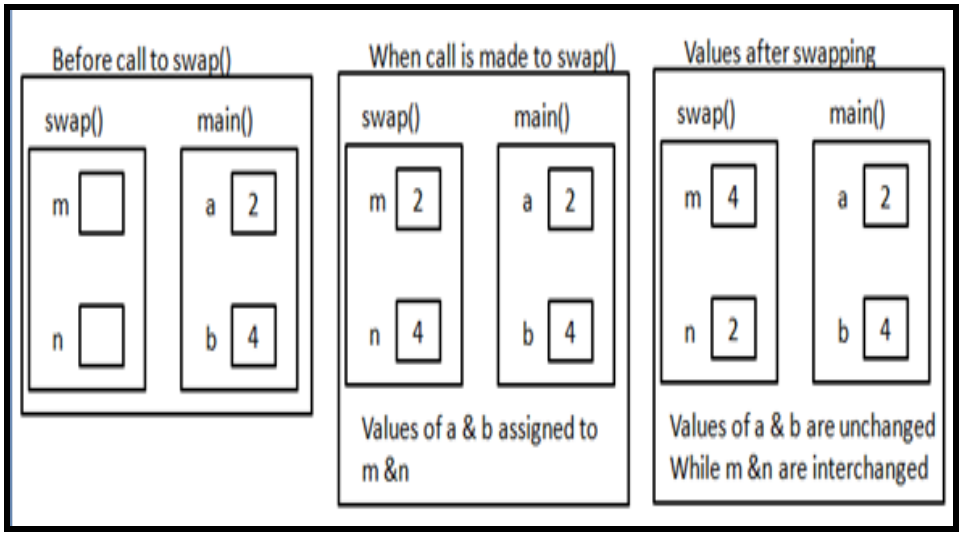
```
Please enter 2 positive integers:        1 4
Values before swapping are (in main ()):
a = 1            b = 4
Values before swapping are (in swap ()):
m = 1            n = 4
Values after swapping are (in swap ()):
m = 4            n = 1
```

```
Values after swapping are (in main ()):
a = 1          b = 4
```

Second Run

```
Please enter 2 positive integers:      11  51
Values before swapping are (in main ()):
a = 11         b = 51
Values before swapping are (in swap ()):
m = 11         n = 51
Values after swapping are (in swap ()):
m = 51         n = 11
Values after swapping are (in main ()):
a = 11         b = 51
```

- The above program swaps the values of two integers taken by the user using a swap function that performs the swapping task.
- The swap function accepts 2 integers from the main function as shown in the prototype.
- Before we do the swapping, we simply print the values of variables a & b so that we may know the state of the variables (i.e. they undergo a change or not).
- The statement swap(a,b); is an example of call by value where the function swap is called by value. Here variables a & b are actual parameters and their values are passed while invoking the swap function.
- The function definition of swap(); shows int m and int n, these are called formal parameters and they receive the values of variables a & b passed from main().
- Inside swap(), before we swap the values of the variables m &n we print their values on the screen. Once the swapping is done the values of the variables m & n are again printed on the screen.
- When the control returns back to the main function, the values of variables a & b are again printed on the screen.
- From the above program the following could be noted:
    1. The values of actual parameters (a & b) are passed to the formal parameters(m &n).
    2. Any change done to the formal parameters do not change the actual arguments as shown in the output.



10.3. CALL BY REFERENCE

- The call by value mechanism is a read only way of communication with the function and it does not change the values of the actual arguments. It makes the functions more self-contained, protecting them against accidental side effects.
- But sometimes there may be situations where a function may need to change the value of the parameter passed to it. This is done by using the call by reference mechanism.
- To pass a parameter by reference instead of by value, we simply append an ampersand, &, to the data type in the functions parameter list which the local variable a reference to the argument passed to it.
- Now the argument is ***read-write*** instead of **read-only** and any change to the local variable inside the function will cause the same change to the argument that was passed to it.
- When parameters are passed by reference, the formal arguments become aliases to the actual arguments in the calling function. This is similar to working with the same original values with two different names.
- **Reference Variable –** A reference variable is an alias or alternate name for a previously defined variable. Later on the two variable names can be interchangeably used to represent the value.

The Syntax to create a reference variable is as follows:

```
data-type & reference-name = variable-name;
```

Example:

```
int  a = 20;
int & b = a;                          //b is a reference      variable
cout<<a<<endl<<cout<<b;    //will both print a value of 20
a=a+10;
cout<<b;                             //will print 30
```

- Example of swapping two numbers using pass by reference

```
/*************************************************
************
Program
Author: Nikhil Pawanikar
Description: Program to swap the values of two numbers (call by
reference)
*************************************************
************/
#include <iostream.h>
#include <conio.h>
void swap(int &,int &);                          //prototype
int main(void)
{
int a,b;
cout << "Please enter 2 positive integers:\t ";cin >> a>>b;
cout<<"\n Values before swapping are (in main ()):\n a = "
<<a<<"\t b = "<<b<<endl;
swap(a,b);                          //call by value, actual arguments
cout<<"\n Values after swapping are (in main ()):\n a = "
 <<a<<"\t b = "<<b<<endl;
getch();
return 0;
}
void swap(int & m,int & n)              //definition,          formal
argument
{
int temp;
cout<<"\n Values before swapping are (in swap ()):\n m =
"<<m<<"\t n = "<<n<<endl;
temp = m;
m = n;
n = temp;
cout<<"\n Values after swapping are(in swap ()):\n m = "
<<m<<"\t n = "<<n<<endl;
```

```
}
```

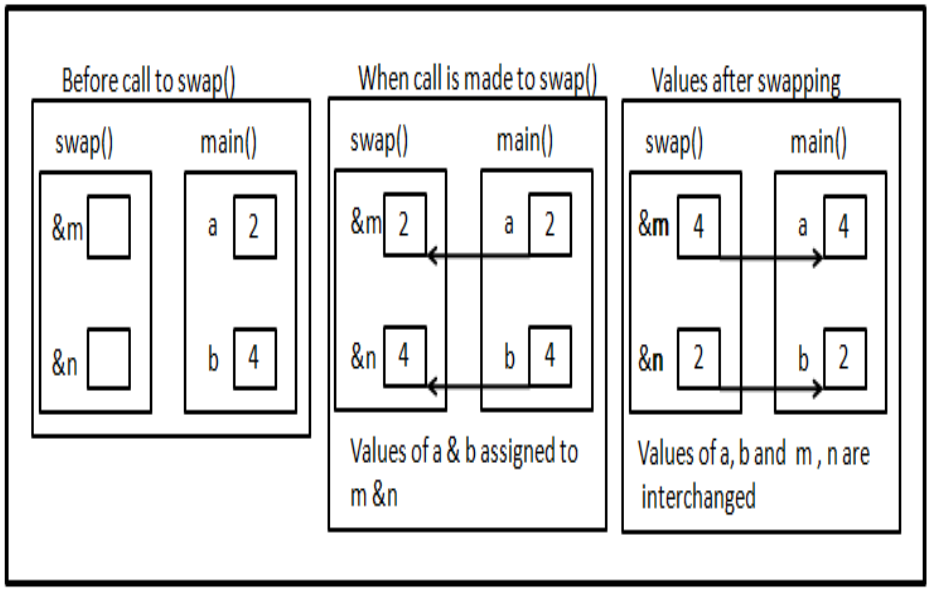
Output:
First Run

```
Please enter 2 positive integers:        1 4
Values before swapping are (in main ()):
a = 1            b = 4
Values before swapping are (in swap ()):
m = 1            n = 4
Values after swapping are (in swap ()):
m = 4            n = 1
Values after swapping are (in main ()):
a = 4            b = 1
```

Second Run

```
Please enter 2 positive integers:        11  51
Values before swapping are (in main ()):
a = 11           b = 51
Values before swapping are (in swap ()):
m = 11           n = 51
Values after swapping are (in swap ()):
m = 51           n = 11
Values after swapping are (in main ()):
a = 51           b = 11
```

The above program can be summarized as follows:



The following can be concluded:

- Using reference variable any changes made to the formal parameters are reflected on the actual parameters since they are simply aliases.

10.4 INLINE FUNCTIONS

- Using functions adds to the overhead of program execution. This overhead involves time and space to invoke the function, passing parameters, allocate memory for variables, store the values of variables in the memory allocated, executing the instruction, returning value to the calling function, etc.
- C++ offers the concept of **inline functions** to address this problem. With inline functions the compiler replaces each call to the function with explicit code for the function .i.e. An inline function is expanded when the function is invoked.
- A function is made inline function by simply adding the keyword inline to the function definition.

Ex.

```
inline int square(int m)
{
        return m*m;
}
```

- Using inline function involves a tradeoff between faster execution and memory being used. A function with many instructions that is called multiple times gets copied every time it is executed and occupies more memory.

10.5 RECURSIVE FUNCTIONS

- In C++ , a recursive function is one which calls itself. It is a function being executed where one of the instructions is to "repeat the process". It sounds similar to a loop.

Ex.

```
void recursive();

int main()
{
recursive();
return                          0;
}

void recursive()
{
        recursive();
}
```

The above function will logically run in an infinite loop.

Example : Program to find the factorial of a number

```
/*************************************************************
**************
Program 9.2
Author: Nikhil Pawanikar
Description: Program to display factorial of a number entered by user by
using recursion
*************************************************************
**************/
#include <iostream.h>
#include <conio.h>

int factorial(int);
int main(void)
{
        int number,fact;
        cout << "Please enter a positive integer: ";
        cin >> number;
        fact=factorial(number);
        cout << number << " factorial is: " << fact << endl;
        getch();
        return 0;
}

int factorial(int number)
{
        int temp;

        if(number <= 1)
        {
              return 1;
        }
        else
        {
        temp = number * factorial(number - 1);
        }
        return temp;
}
```

Output:

First Run

```
Please enter a positive integer:        4
4 factorial is: 24
```

Second Run

```
Please enter a positive integer:        0
0 factorial is: 1
```

10.6. REVIEW QUESTIONS

1. Explain Call by value
2. Explain Call by reference
3. Explain the difference between call by value and call by reference
4. Write short notes on:
   a. Inline functions
   b. Recursion
5. Write a program to swap two numbers without using a third variable using call by reference.

10.7. REFERENCES & FURTHER READING

1. Let us C – Yashwant Kanetkar, Chapter 5.
2. Object Oriented Programming with C++ - E. Balaguruswamy, Chapter 4
3. Programming in C++, Schaums Outlines – John R. Hubbard, Chapter 5.

Solved Example:

Program to swap two numbers without using a third variable using call by reference.

```
#include <iostream.h>
#include <conio.h>

void swap(int &,int &);
int main(void)
{
        int a,b;
        clrscr();
        cout << "Please enter 2 positive integers: ";
        cin >> a>>b;

        cout<<"\n Values before swapping are (in main ()):\n a
= "
            <<a<<"\t b = "<<b<<endl;
```

```
        swap(a,b);                                          //call     by
value

        cout<<"\n Values after swapping are (in main ()):\n a =
"
           <<a<<"\t b = "<<b<<endl;

        getch();
        return 0;
}
void swap(int & m,int & n)
{
        m = m + n;
        n = m - n;
        m = m - n;
}
```

Output:

First Run

```
Please enter 2 positive integers:        1 4
Values before swapping are (in main ()):
a = 1            b = 4
Values after swapping are (in main ()):
a = 4            b = 1
```

Second Run

```
Please enter 2 positive integers:        11  51
Values before swapping are (in main ()):
a = 11           b = 51
Values after swapping are (in main ()):
a = 51           b = 11
```

}

11

Chapter 11
Derived Data types (Arrays, functions)

Unit Structure

11.0 Objectives
11.1 Introduction to arrays,
11.2 2-D arrays,
11.3 Multidimensional arrays,
11.4 Arrays in functions,

11.1 INTRODUCTION TO ARRAYS

What is array?

An array is an groups of elements that can be identifies as a similar types.i.e array of integer types ,array of floating types , array of Character types and so on.

Array can be categorized into two part

1) Single dimensional Array
2) Multi dimensional Array

How we declared an Array?

Just like declaration as ordinary variable of different types such as int rollno, float avg, char name and so on,

As we mentioned in the definition of array , Array is an groups of elements that can be identifies as a similar types. Yet there is difference between ordinary variable and array variable that is you needs to tell the compiler what kind of array is you are defining, an array of books? An array of students? An arrays of cloths? because the compiler wants to Know that how much amount of space will be required to stored an array element or data item in the computer memory. When declaring the array of elements in the program, the compiler will put each item of array in appropriate location

Like any other variable, The syntax of declaration of an array is:

Datatypes variable_name[dimension/size]

Here datatypes could be an int, a float , an char etc,
Variable_name is following according to C++ naming rules ,After the Variable_name is then, we have to specifies the dimension or size of an array(i.e the size should be specifeid with the opening square and closing square[]).

Some example of an declaring an Arrays.
int rollno[20];
char grade[100] ;
double marks[20];

int rollno[20]; declares a group of element or array of 20 values, each element is being an integer.

char grade[100]; declares an array of 100 charcater values.

double marks[360]; declares an array of double-precision numbers. There are 360 of these items in the group.

int rollno[20];

rollno is an array of 20 integer and each elements of an array are accessible by the superscript(index) and if an array consist n element of similar datatype ,then the array ranges is starting from 0 to n-1 because the starting index of an array is 0 and ending is n-1.

Consider the above Example is: int rollno[20]; means the starting index of rollno array is 0(known as lower bound) and ended with an last index is 19(known as Upper bound)

Initializing an Array:

We can individually assigned value to an array such as

Rollno[0]=1; rollno[1]=2;……..upto rolln0[19]=20;

Just like any variable can be initialized, an array also can be initialized. To accomplish this, for a one-dimensional array, the syntax used is:

DataType variable_Name[dimension/size] = { element1, element2, .., element n};

the datatype specify that what kind of array you are declaring, then followed by the array name, and the square brackets. After specifying the dimension or not, and after the closing square bracket, type the assignment operator. The elements, also called items that compose the array are included between an opening curly brace '{' and a closing curly brace '}'.

Each element of an array is separate from the other by a comma operator. After the bracing is complete then you end it with a semi-colon.
Consider an example:
int rollno[12]={1,2,3,4,5,6,7,8,9,10,11,12}
double distance[5] = {44.14, 720.52, 96.08, 468.78, 6.28};

If you have decided to initialize the array while you are declaring it, you can omit the dimension. Therefore, these arrays can be declared as follows:

int rollno[]={1,2,3,4,5,6,7,8,9,10,11,12}
double distance[] = {44.14, 720.52, 96.08, 468.78, 6.28};

Example:

Once you have initialized an array's elements, you can display its elements or items of array using cout. Here is an example:

```
#include <iostream.h>
int main()
{
     int rollno[] = {101, 102, 103, 104, 105};

   cout << "2nd member = " << rollno[1] <<endl;
   cout << "5th member = " << roll[4] << endl;

     return 0;
}
```

The result would produce:

```
2nd member = 102
5th member = 105
```

Using this technique, each element or item of the array can be accessed. Here is an example:

```
#include <iostream.h>
int main()
{
      int rollno[] = {101, 102, 103, 104, 105};

      cout << " rollno 1: " << rollno[0] << endl;
      cout << " rollno 2: " << rollno[1] << endl;
      cout << " rollno 3: " << rollno[2] << endl;
      cout << " rollno 4: " << rollno[3] << endl;
      cout << " rollno 5: " << rollno[4] << endl;
```

```
      return 0;
}
```

The result would produce:

```
rollno 1: 101
rollno 2: 102
rollno 3: 103
rollno 4: 104
rollno 5: 105
```

The Size of an Array:

- When array has been declared, the programmer has to decides that how many element should have to initialize depend the size of array. The size of array decrease or increases depend upon the requirement of program.
- Sometimes the programmer doesn't want to increase or decreases the size of an array, One method is fixed the size of an array i.e constant , const is reserved keyword or Qualifier , if we declared the const keyword with an variable assigned with certain value that value cannot modified or altered during the execution of the program.
- If the program is long and the array is declared in some unusual place, this could take some time. The alternative is to define a constant keyword before declaring the array and use that constant to hold the dimension or size of the an array. Here is an example:

```
#include <iostream.h>
int main()
{
const int number_Of_Items = 5;
int  rollno[number_Of_Items]={101,  102,  103,104,
105};

    cout << " rollno 1: " << ro11no[0] << endl;
    cout << " rollno 2: " << rollno[1] << endl;
    cout << " rollno 3: " << rollno[2] << endl;
    cout << " rollno 4: " << rollno[3] << endl;
    cout << " rollno 5: " << rollno[4] << endl;

    return 0;

    }
```

The result would produce:

```
rollno 1: 101
rollno 2: 102
```

```
rollno 3: 103
rollno 4: 104
rollno 5: 105
```

You can use such a constant in a **for** loop to scan the array and access each elements of an array. Here is an example:

```
#include <iostream.h>
int main()
{
   const int numberOfItems = 5;
   introllno[numberOfItems]={101,102,103,104,105};
   cout << "Members of the array\n";
   for(int i = 0; i < numberOfItems; ++i)
       cout << "rollno" << i + 1 << ": " <<
rollno[i] << endl;

       return 0;
}
```

In both cases, this would produce:

```
Members of the array

rollno 1: 101
rollno 2: 102
rollno 3: 103
rollno 4: 104
rollno 5: 105
```

We know that size of an array, that we can easily count the number of element in given array , as the size of array is small , Consider the situation where array size is long for example 300 or 500 , you wouldn't start counting the number of members. The C++ provides the sizeof()operator that can be used to get the dimension of an array. The syntax you would use is:

sizeof(*ArrayName*) / sizeof(*DataType*)

Imagine you have declare an array as follows:

int number[] = {18, 42, 25, 12, 34, 15, 63, 72, 92, 26, 26, 12, 127, 4762, 823, 236, 84, 5};

Instead of counting the number of elements of this array, you can use the

sizeof() operator as follows:

int Number_OfItems_Of_TheArray = sizeof(Number)/sizeof(int);

Advantage of using the **sizeof** operator is used to get the number of elements of the array is that it can be used on a for loop to traverse from an array, either to locate the element or to look for a value in the array. Here is an example of using this concept:

```
#include <iostream.h>
int main()
{
    int rollno[] = { 101, 102, 103, 104, 105};
    // Using the sizeof operator to get the size
of the an array
    Int num = sizeof(distance) / sizeof(double);
    cout << "Array members and their values\n";
    // Using a for loop to scan an array
    for(int i = 0; i < num; ++i)
       cout << "rollno : " << i + 1 << rollno[i]
<< endl;
    return 0;
}
```

This would produce:

```
Array members and their values

rollno 1: 101
rollno 2: 102
rollno 3: 103
rollno 4: 104
rollno 5: 105
```

11.2 2-D ARRAYS

As we have seen the example of one- dimensional array, we also have seen the how to initialized the data item / elements to one – dimensional array

The array thats contain numbers of rows and columns like a the matrix i.e. the array which contain the two square brackets and mentioned the size of an arrays is called as a 2-D arrays or 2 dimensional arrays

Declaration of 2-D arrays:

Datatypes variablename[dimension/size][dimension/size]

int anArray[3][5]; // a 3-element array of 5-element arrays or three rows and 5 columns

In this case, since we have 2 subscripts or index , this is called as a two-dimensional array. In a two-dimensional array, the first subscript or index consider as an row, and the 2nd subscript or index considers as an column. the above two-dimensional array is laid out as follows:

[0][0]	[0][1]	[0][2]	[0][3]	[0][4]
[1][0]	[1][1]	[1][2]	[1][3]	[1][4]
[2][0]	[2][1]	[2][2]	[2][3]	[2][4]

A 2-dimensional array declaration defines as like of matrix of variables or the same type:

int anArray[3][5]

anArray

anArray [0][0]	anArray [0][1]	anArray [0][2]	anArray [0][3]	anArray [0][4]
anArray [1][0]	anArray [1][1]	anArray [1][2]	anArray [1][3]	anArray [1][4]
anArray [2][0]	anArray [2][1]	anArray [2][2]	anArray [2][3]	anArray [2][4]

Since computer memory is linear, the elements of a 2-dimensional array are actually stored linear formed
int anArray[3][4]

anArray [0][0]	anArray [0][1]	anArray [0][2]	anArray [0][3]	anArray [0][4]	anArray [1][0]

- These variables can be referred to individually with the subscript operator, e.g., anArray[1][2].
- The set of variables can be referred to as an aggregate with the array name, e.g., anArray.
- In C++, the first element of an array has the subscript 0.
- In C++, the last element of an array with n element has the subscript n-1.
- In C++, the range of valid index of an m x n array is [0..m-1[[0..n-1].
- Using a index outside of the valid range is an error. This error will not be detected by the compiler; it will shown as a run time error. In some cases, it will cause the program to crash; in other cases, the

program will appear to run normally, but will produce incorrect results.

- The subscript of an array is also called an index.
- A two-dimensional array can called as an array of arrays.

Initialization:

Example of initialization:

```
  int anArray [2] [3] = { {1, 2, 3}, {4, 5, 6} };

or
  int anArray [] [] = { {1, 2, 3}, {4, 5, 6} };

or
  int anArray[] [] = { {1, 2, 3}, //row 0
                       {4, 5, 6} };//row 1
```

Example of a two-dimensional array:

```
#include <iostream.h>
#define nNumRows      10
#define nNumCols      10

void main()
{
    int nProduct[nNumRows ][nNumCols ] = { 0};
    // Calculate a multiplication table
    for (int nRow = 0; nRow < nNumRows; nRow++)
       for (int nCol = 0; nCol < nNumCols; nCol++)
           nProduct[nRow][nCol] = nRow * nCol;

    // Print the table
    for (int nRow = 1; nRow < nNumRows; nRow++)
    {
       for (int nCol = 1; nCol < nNumCols; nCol++)
           cout << nProduct[nRow][nCol] << "\t";

       cout << endl;
    }
```

This program calculates and prints a multiplication table for all values between 1 and 9. Note that when printing the table, the for loops start from 1 instead of 0. This is to omit printing the 0 column and 0 row, which would just be a bunch of 0s! Here is the output:

1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45
6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81

Example:

Write your own C++ program that transposes matrix. Program stores given matrix dimensions and every single matrix element must be given. Transposed matrix is the one with rows and columns switched.

```
#include <iostream.h>
#define ROW      50
#define COL      50

int main()
{

int i, j, m, n, temp;
int mat[ROW][COL];

  // variable dim  is set to smaller value of defi
ned
  // maximal number of rows and columns

    int dim = (ROW < COL)? ROW : COL;

    // storing matrix size

    do {

        cout<<"Input number of rows”<< dim;
        cin>>m;
        cout<<"Input number of columns” << dim;
        cin>>n;

    }while (m < 1 || m > dim || n < 1 || n > dim);

    // storing matrix elements

    cout<<"\nInput of matrix elements :\n";
```

```
   for (i = 0; i < m; i++) {
      for (j = 0; j < n; j++) {

          cout<< i<< j;
            cin>>mat[i][j];

        }
    }

    // printing matrix before transposing

    cout<<"\n\nMatrix before transposing:\n";
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {

         cout<< mat[i][j];

      }
        cout<<"\n";
    }

   // Tranpose Matrix

    for ( i=0; i<m; ++i ) {
   // second loop must start from i+1.
        for ( j=i+1; j<n; ++j ) {

                temp = mat[i][j];
                 mat[i][j] = mat[j][i];
                 mat[j][i] = temp;
        }
    }


  // print after transposing
   //i.e number of rows becomes number of columns

    cout<<"\nMatrix after transposing:\n";
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {

              cout<< mat[i][j];
        }
        cout<<"\n";
    }
} // main
```

Example of program's execution:

```
Input number of rows < 50: 3
Input number of columns < 50: 2
Input of matrix elements :
Input element [0][0] : 1
Input element [0][1] : 2
Input element [1][0] : 3
Input element [1][1] : 4
Input element [2][0] : 5
Input element [2][1] : 6

Matrix before transposing:

1      2
3      4
5      6

Matrix after transposing:

1      3      5
2      4      6
```

11.3 MULTIDIMENSIONAL ARRAYS

The elements of an array can be of any data type, An array of arrays is called a **multidimensional arrays**.

Multidimensional arrays may be larger than two dimensions. Here is a declaration of a three-dimensional array:

```
int anArray[5][4][3];
```

11.4 ARRAYS IN FUNCTIONS

Array variables as parameters:

When an array is passed as a parameter, only the memory address of the an array is passed (but not the values of the variable which is assigned to arrays index variable). An array as a parameter is declared similarly to an array as a variable, but not size of an array (no limit)are specified. The function doesnot know how much memory space is allocated for an array.

Example -- Function to add numbers in an array:

This main program calls a function to add all the elements in an array and uses the returned value to compute the average.

```
#include <iostream.h>

float addition(const float y[], const int size);
                                        / /prototype

   int main() {
   float a[1000]; // Declare an array of 1000 floats
   int n = 0;     // count the number of values in a.

    while (cin >> a[n]) {
        n++;
    }

    cout << "Average = " << addition(a, n)/n << endl;

    return 0;
}

// sum adds the values of the array it is passed.
float addition(const float y[], const int size) {
    float total = 0.0;  // the sum is accumulated
    for (int i=0; i<size; i++) {
        total = total + x[i];
    }
    return total;
}
```

12

Chapter 12
Introduction to pointers

Unit Structure:

12.1 INTRODUCTION TO POINTERS

When you declare the variables in the program the complier allocated a logical address to the variable in the main memory i.e When you declare a variable in the progaram, the computer allocates amount of space for that variable, and uses the variable's name to refer to that memory space.

When you declares an a varaibles in the program the space is created in the memory that assigning the address of the particular variable that space is used for holding the values of that particular variable. Therefore, everything you declare has an address, just like the address of your house. You can find out what address a particular variable is using. when you declares a variable,its tell the compiler what kind of variable is .i.e which data type variable has in the program i.e whether the variable consist the integer, floating point character and so on.

12.1 DEFINITION OF POINTER

Pointer is an variable is used to hold the memory address of another variable.

if you want to see variable's address of a particular variable, you can use the &(ampersand) operator followed by the name of the variable.

Let consider an example for accesing the memory address of the variable numOfStudents. As if you decaled the variable like

int numOfStudents;
you can get the address of variable numOfStudents and where the variable laocation is loacted by using:
cout << &numOfStudents;

This program would give you the address of the declared variable:

```
#include <iostream.h>
int main()
{

  int num;

  cout << " the address of variable num is:     <<
&num;

   cout << "\n\n";

    return 0;

}
```

After executing the program, you could get:

```
The address  of variable num at: 0x0065FDF4
```

Notes: The address of variable will differ when the program is execute on different computers .

Here the variable address in Hexadecimal format.

12.2 Why Use Pointers?

When you declare an variable the operating system assisgned the particular unique address to the variable in the main memory, as we compile the program, the compiler assigned the logical address to the memory, here the content which is stored in the logical address space is modified through the pointer variable.

Note that the pointer is an variable which also has the logical address located in the memory

Consider the example pass by reference, you can pass an argument to a function, the argument is passed using its address. This allows the calling function to find the address of the variable (the argument) and athat can be used to acesss the value directly. This allows the calling function to alter the real value of the argument.

Applying this concept, a pointer can allow you to return many values from a function; as opposite to regular argument passing(pass by a value) where the data changed inside of the called function, when the execution of the functin is over, the calling function regain its previous value . Therefore, passing arguments as pointers allows a function to return many values, even if a function is declared as void.

When you declare an array, you must specify the dimension of the array. what if you don't know and don't want to know the dimension of the array? Pointers provide an ability that regular arrays do not have. Since pointers provides the better management system of memory, a pointer can store an array of almost any size;

Using this feature, when declaring a pointer instead of an array, you do not have to worry about the size of the array, the compiler will take care of that.This feature also allows you to pass pointers to a function (just like arrays) and return a value that has been altered even if the function is declared as void. This is like a dynamic with multidimensional arrays.

Just any other ordinary variable in C++, you should declare and initialize a pointer variable before using it. To declare a pointer variable, use an identifier i.e. datatype , followed by an asterisk (*), followed by the name of the pointer, and a semi-colon.

The Syntax for declaration of Pointer
DataType * Pointer_Name;

Pointer variable should be an int, a char, a double, etc. The identifier should be the same type of identifier the pointer variable will point to. Therefore, if you are declaring a pointer that will point to an integer variable, the pointer identifier should be an integer.

The asterisk (*) that knows the compiler that the variable has be declared is a pointer that point to an respective datatype. There are three ways you can type the asterisk. These are

DataType* Pointer_Name;

DataType * Pointer_Name;

DataType *pointer_Name

Since the name of the pointer is indeed the name of a variable, you will follow the naming rules that govern every C++ variable.

```
#include <iostream.h>
int main()
{

    int num;
    int *ptr;
    cout << "The address of num is : " << &num <<
    cout << "The address of pointer is: " << &ptr
    cout << "\n\n";
    return 0;
}
```

After executing the program, you might get the result:

```
The address of num is : 0x0065FDF4
The address of pointer is : 0x0065FDF0
```

12.3 INITIALIZING A POINTER

As we have already seen that , a variable should be initialized before being used. This allows the compiler to put value into the memory space allocated for that variable.

To use a pointer varaible ptr . You need to tell the compiler that pointer ptr will be used to point to the address of variable X . Do this you should have to initialized the pointer variable by some value. A pointer is initialized like an ordinary variable, by using the assignment operator (=).

There are two main ways you can initialize a pointer. When declaring a pointer like this:

int* Ptr;

initialize it by following the assignment operator with & operator and the name of the variable, like this

int* Ptr = &Variable;

This program could also have the pointer initialized as:

```
#include <iostream.h>
     int main(){
int num = 12;
```

```
int *ptr = &num;
cout << "The value of num is: " << num << "\n";
cout << "The value of pointer variable is: " <<
*ptr;
cout << "\n\n";
return 0;

}
```

```
    The program would produce:
    The value of num is: 12
    The value of pointer variable is: 12
```

Another of the program, you can first declare both variables, then initialize them later on,when you needed.

```
#include <iostream.h>
int main()
{

int num;
int *ptr;
ptr = &num;
num = 23;
cout << " The value of num is: " << num << "\n";
cout <<  The value of pointer variable  is:= " <<
*ptr<< "\n";
cout << "\n";
return 0;

}
```

The program would produce:

```
The value of num is: 23
The value of pointer variable is: 23
```

Once you have declare a variable and assign it to a pointer, during the course of your program, the value of a variable is likely to change, you can therefore assign it a different value:

```
#include <iostream.h>
int main()
{
    int num;
    int *ptr;
    ptr = &num;
```

```
    num = 23;
    cout << " The value of num is: " << num <<
                                        "\n";
    cout << " The value of pointer variable is:
                          " << *ptr << "\n";
    num = 35;
    cout << " The value of num is: " << num <<
                                        "\n";
    cout << " The value of pointer variable is:
                          " << *ptr << "\n";
    cout << "\n";
    return 0;

}
```

```
The value of num is: 23

The value of pointer variable is: 23

The value of num is: 35

The value of pointer variable is: 35
```

Both *ptr and num have the same value. This allows you to change the value of the pointer directly and affect the main variable meanwhile. Therefore, you can safely change the value of the pointer and it will be assigned accordingly.

To see an example, make the following change to the file:

```
#include <iostream>
int main()
{
 int num;
 int *ptr;
 ptr = &num;
 num = 26;
 cout << " The value of num is:" << num <<
"\n";
 cout << " The value of Pointer variable is:
" << *ptr << "\n";
 num = 35;
 cout << " The value of num is:" << num <<
"\n";
 cout << " The value of Pointer variable is: " <<
*ptr << "\n";
 *ptr = 144;
```

```
cout << " The value of num is: " << num <<
"\n";
cout << " The value of Pointer Variable is: " <<
*ptr << "\n";
cout << "\n";
    return 0;
}
```

This would produce:

```
The value of num is: = 26
The value of Pointer Variable is: = 26
The value of num is:  = 35
The value of Pointer Variable is: = 35
The value of num is:  = 144
The value of Pointer Variable is: = 14
```

12.4 A POINTER TO A POINTER

In this program, you can declare a new variable that is a pointer variable that itself points to another pointer. When you declaring such a variable, precede it with two *. Sign.After declaring the pointer, before using it, you must initialize it with a reference to a pointer, that is, a reference to a variable that was declared as a pointer. Here is an example:

```
#include <iostream.h>
int main()
{
     int num = 26;
int *ptr;
int **ptrToPtr;
ptr = &num;
ptrToPtr = &pointer;
cout << " The value of num is: = " << num <<
"\n";
cout << "  The value of pointer variable is: = "
<< *ptr << "\n";
cout << " The value of pointer to pointer is: = "
<< **ptrToPtr << "\n";
return 0;
}
```

This would produce:

```
The value of num is: = 26
The value of Pointer variable is: = 26
The value of Pointerto Pointer Variable is: = 26
```

After initializing a pointer, if you change the value of the variable that's it points to, the pointer value woud be change. Consider the following program:

```
#include <iostream.h>
int main()
{
 int num= 26;
 int *ptr;
 int **ptrToPtr;
 ptr = &num;
 ptrToPtr = &ptr
 cout << "  The value of num is:    = " << num << "\n";

 cout << " The value of Pointer variable is: = " << *ptr <<
"\n";
           cout << "The value of Pointer to Pointer is: = "
<< **ptrToPtr << "\n";
 num = 4805;
 cout << "After changing the value of the main variable...\n";
 cout << " The value of num is:    = " << num << "\n";
 cout << " The value of Pointer Variable is:  = " << *ptr <<
"\n";
 cout << " The value of Pointer to Pointer variable is:  = "
<< **ptrToPtr<<"\n";
 return 0;
}
```

This would produce:

```
The value of num is:   = 26
The value of Pointer Variable is:  = 26
The value of PointertoPointer Variable is:  = 26
After changing the value of the main variable...
The value of num is:   = 4805
The value of Pointer Variable is:  = 4805
The value of PointertoPointer Variable is:  =4805
```

12.5 OPERATIONS ON POINTERS

A variable has a value that supposed to change from time to time. Since a pointer is a variable whose value points to another variable, the value of a pointer is affected or updated by the variable it points to. You can use indirection operator to change the value of a pointer when changing its main variable.

To get a value from the user by using the **cin** operator. When using a pointer to get a value from the user, please don't forget the * operator, otherwise, the compiler would get confused.

We have already know how to get the value from the user and display the value of a regular variable from the user:

```
#include <iostream.h>
int main()
{

       int students;
       cout << "Number of students: "
       cin >> students;
       cout << "\nNumber of students: " <<
       students;
       cout << "\n\n";
       return 0;

}
```

Once you have got a value from the user and store it in a variable, it is available:

```
#include <iostream.h>
int main()
{

       int students;
       int *ptrstudents;
       ptrstudents = &students;
       cout << "Number of students: ";
       cin >> students;
       cout << "\nNumber of students: " <<
       students << "\nThat is: " <<
*ptrstudents << “students.";
                cout << "\n\n";
                 return 0;

}
```

This could produce:

```
Number of students: 24
Number of sudents: 24
That is: 24 students
```

In the same way, you can request a value from the user and store it in the pointer. To see an example, make the following change to the file:

```
#include <iostream.h>
int main()
{

 int students;
 int *ptrstudents;
 ptrstudents =&students;
 cout << "Number of students ";
 cin >> *ptrstudents;
 cout << "\nNumber of students: " << students
 << "\nThat is: " <<
*ptrstudents << students.";
            cout << "\n\n";
 return 0;
}
```

You can use various pointers on the same program. Apply an example by making the following changes:

```
#include <iostream.h>
int main()
{
            int maleTeacher;
            int femaleTeacher;
 int *ptrmaleTeacher;
 int *ptrfemaleTeacher;
 ptrmaleTeacher = &maleTeacher;
 ptrfemaleTeacher = &femaleTeacher;
 cout << "Number of male Teacher : ";
 cin >> *ptrmaleTeacher;
 cout << "Number of female Teacher : ";
 cin >> *ptrfemaleTeacher;
 cout << "\nNumber of Teachers:";
 cout << "\n male Teacher:" << "\t" <<
maleTeacher<< "\nThat is: " << *ptrmaleTeacher <<
" Teachers.";
 cout << "\nfemale Teacher:" << "\t" <
maleTeacher<< "\nThat is: " << *ptrfemaleTeacher
<< " Teachers.";
   cout << "\n\n";
 return 0;
}
```

We have learned how to perform algebraic calculations and expressions in C++. When performing these operations on pointers,

remember to use the * for each pointer involved. The calculations should be as smooth.

```
#include <iostream.h>
int main()

{
       int maleTeacher;
       int femaleTeacher;
       int totalTeacher;
       int *ptrmaleTeacher;
       int *ptrfemaleTeacher;
       int *ptrTotalTeacher;
       ptrmale = &maleTeacher;
       ptrfemale = &femaleTeacher;
       ptrTotal = &totalTeacher;
       cout << "Number of male  Teachers: ";
       cin >> *ptrmaleTeacher;
              cout << "Number of female Teachers:
";
       cin >> *ptrfemaleTeacher;
       cout << "\nNumber of Teacher:";
       cout    <<    "\nMaleTeacher:"    <<    "\t"
<<maleTeacher
                <<     "\nThat     is:     "     <<
*ptrmaleTeacher << " Teachers.";
               cout << "\nfemaleTeacher:" << "\t"
<< femaleTeacher
                <<     "\nThat     is:     "     <<
*ptrfemaleTeacher << " Teachers.";
       Total =male + female;
       *ptrTotalTeacher    =    *ptrmaleTeacher    +
*ptrfemaleTeacher;
       cout << "\n\nTotal number of Teachers: " <<
totalTeacher;
       cout << "\nThere are " << *ptrTotalTeacher
<< " Teachers";
       cout << "\n\n";
       return 0;

}
```

This would produce:

```
Number of male Teachers: 26
Number of female Teachers: 24
maleTeacher: 26
```

```
That is: 26 Teachers
femaleTeacher: 24
That is: 24 Teachers
Total number of Teachers: 50
There are 50 Teachers
```

12.6 POINTERS IN FUNCTION

We know that a function uses arguments in order to carry its operation. The arguments value are usually provided to the function. When necessary, a function also declares its own variable to get the value. Like other variables, pointers can be provided to a function,

When declaring a function that takes a pointer as an argument, you have to use the asterisk for the argument or for each argument(formal arguments). When calling the function, use the references to the variables. The function will perform its assignment on the referenced variable(s). After the function has performed its assignment, modifies value(s) of the argument(s) will be stored and given those modified value to the calling function(actual arguments).

Here is a starting file from what we have learned so far:

```
#include <iostream.h>
int main()
{
 int a = 12;
            int b = 5;
            cout << "The value of a = " << a <<
endl;
 cout << "The value of b = " << b << endl;
 cout << endl;
 return 0;
}
```

This would produce:

```
The value of a = 12
The value of b = 5
```

To pass arguments to a function, you can make the following changes:

```
#include <iostream.h>
int main()
{
 int a = 3;
 int b = 5;
```

```
void  Value(int sa, int pa);
cout << "When starting, within main():\n";
cout << "\t The value of a = " << a<< endl;
cout << "\t The value of b  = " << b << endl
Value(sa, pa);
cout  <<  "\n\nAfter  calling    ChangeValue()  ,
within main():\n";
cout << "\t The value of a = " << a<< endl;
cout << "\t The value of b  = " << b << endl
cout << endl;
return 0;
}

void  Value (int s, int p)
{
s = 8;
p = 12;
cout  <<  "Within  ChangeValue()"<<  "\n\tThe  value
of s = " << s<< "\n\tThe value ofp = " << p;
}
```

After executing, the program would produce:

```
When starting, within main():

        The value of a = 3
        The value of b =  5

Within Value()

        The value of s = 3
        The value of p =  5
After Within Value(), Within main():

       The value of a = 3
        The value of b =  5
```

To pass pointer arguments, use the asterisks when declaring the function, and use the ampersand **&** when calling the function. Here is an example:

```
#include <iostream.h>
int main()
{
       int a = 12;
       int b = 5;
       void Value(int sa, int pa);
       void ChangeValue(int *sa, int *pa);
       cout << "When starting, within main():\n";
```

```
        cout << "\t The value of a = " << a<<
        endl;
        cout << "\t The value of b  = " << b <<
        Value( sa, pa);
        cout << "\n\nAfter calling Value(), within
main():\n";
        ccout << "\t The value of a = " << a<<
        cout << "\t The value of b  = " << b <<
endl
        ChangeValue(&sa, &pa);
        cout << "\n\nAfter calling ChangeValue(),
within main():\n";
        cout << "\t The value of a = " << a<<
endl;
        cout << "\t The value of b  = " << b <<
endl
        cout << endl;
        return 0;
}
void Value(int s, int p)
{
    s = 8;   p = 5;
    cout << "\nWithin Value()"
        << "\n\t The value of s= " << s
         << "\n\t The value of p  = " << p;
}
void ChangeValue(int *sa, int *pa)
{
    *sa= 26;
    *pa      = 17;
    cout << "\nWithin ChangeValue()"

         << "\n\tThe value of sa = " << *sa
        << "\n\tThe value of pa  = " << *pa;
}
```

The result of executing the program is:

```
When starting, within main():

         The value of a = 12
         The value of b   = 5

Within Value()

         The value of s = 8
         The value of p= 5
```

```
After calling Value(), within main():

        The value of a = 12
         The value of b   = 5

Within ChangeValue()

         The value of sa = 26
         The value of pa   = 17

After  ChangeValue (), within main():

         The value of sa = 17
         The value of pa   = 26
```

12.7 VOID POINTERS

Pointer to Void:

General Syntax:

void* pointer_variable;

Void is used as a keyword

We know that the data type the pointer variable defines is the same as the data type the pointer points to. The address placed in a pointer must have the same type as the pointer.

For example:

Sample Code

```
1. int i;
2. float f;
3. int* abc;
4. float* xyz;
5. then
6. abc=&i;
```

It is correct because the address of integer variable is stored in an integer pointer.

If a user writes the statement:

abc=&f;

The above statement result an error. The address of the floating variable(i.e &f) is stored in an integer pointer (abc) that is incorrect. Similarly, if the programmer tries to place the address of an integer(i.e &i) variable to a float pointer, such as:

xyz=&i;

The above statement will also show an error.

The **Pointer to Void** is a special type of pointer that the programmer can use this variables to point to any data type.

Using the above example, the programmer declares pointer to void :

void*num;

Using the above example's definition and assigning the pointer to void to the address of an integer variable is perfectly correct.

num=&i;

Using the above example to define the pointer to void and assign the pointer to void to the address of a float variable as below is also perfectly correct.

num=&f;

Pointer to void, or a void pointer, is a special type of pointer that has a provides facility of pointing to any data type.

There are limitations in the usage of void pointers that are explained below.

The programmer must note that void pointers cannot be de-referenced in the same manner.

Direct dereferencing of void pointer is not permitted.

The programmer must change the pointer to void as any other pointer type that points to valid data types such as, int, char, float and then dereference it.

This conversion of pointer to some other valid data type is achieved by using the concept of type-casting.

12.8 POINTER TO CONSTANT

Just like an ordinary variable, pointer can declared as constant

To declare a const pointer, use the const keyword between asterisk and the pointer variable

```
 int value=7;
int *const ptr = &value;
```

Just like a normal const variable, a const pointer must be initialized to a value when its declared and its value cannot be changed.This mean that a const pointer is point to same value. In the above example ptr is always point to the address of the value variable. However the value which is declared is still the non-const variable , so we can still the change the value of the **value** variable being pointed via the dereferencing the pointer. i.e

```
*ptr=8;// since its allowed, because ptr is point to non-const variable.
```

It is also possible to declare a pointer to a constant variable by using the const before the data type.

```
int Value = 7;
const int *Ptr = &Value;
```

Note that the pointer to a constant variable does not actually have to point to a constant variable. Instead, think of it this way: a pointer to a constant variable treats the variable as constant when it is accessed through the pointer.

```
Value = 8; // Value is non-const
```

Since it is okay,

But the following is not:

```
*Ptr = 8; // Ptr treats its value as const
```

Because a pointer to a const value is a non-const pointer, the pointer can be redirected to point at other values:

```
int Value = 5;
int Value2 = 6;

const int *Ptr = &Value;
Ptr = &Value2; // okay
```

To summarize:

A non-const pointer can be redirected to point to other addresses.

A const pointer always points to the same address, and this address can not be changed.

A pointer to a non-const value can change the value it is pointing to.

A pointer to a const value treats the value as const (even if it is not), and thus can not change the value it is pointing to.

Finally, it is possible to declare a const pointer to a const value:

```
const int Value;
const int *const Ptr = &Value;
```

A const pointer to a const value cannot be redirected to point to another address, nor can the value it is pointing to be changed.

12.9 CONSTANT POINTERS

When you need to define a constant pointer to a variable/object; for instance, when taking a function address, or when you want to protect a pointer from unintended modifications such as assignment of new address, pointer arithmetic, etc. In fact, an objects this is a constpointer. A constant pointer is declared:

```
 int a= 10;
int *const b = &a;      //b is a constant pointer to an int
*b = 20;                //OK, a is assigned a new value
b++;                    //Error; cannot change conpi
```

const defines a constant pointer, whereas a const variable is declared like this:

```
 const int k = 10; //k's value may not be changed
```

And a const pointer to a const variable:

```
 int *const b = &k; //b  is a constant pointer to a const int
*b = 20; //Error; k's value cannot be modified
b++; //Error; cannot modify a const pointer
```

12.10 GENERIC POINTER

The **void pointer**, also known as the generic pointer, is a special type of pointer that can be pointed at objects of any data type. A void pointer is declared like a normal pointer, using the void keyword as the pointer's type:

When a variable is declared as being a pointer to type **void** it is known as a generic pointer. Since you cannot have a variable of type **void**, the pointer will not point to any data and therefore cannot be dereferenced.

It is still a pointer though, to use it you just have to cast it to another kind of pointer first. Hence the term Generic pointer.

This is very useful when you want a pointer to point to data of different types at different times.

Here is some code using a void pointer:

```
#include <iostream.h>
int main()
{
    int  num[3] = {10,20,30};
    char name[30] = "Welcome to C World";
    int *pint = NULL;
    void *pvoid = NULL;
    int i;
    pint = &num;
    for ( i=0; i<3; i++ )
     cout<<*(pint + i );
     cout<<"\n";

     // The same can be done using void pointer as
follows.
     pvoid = &num;
    for ( i=0; i<3; i++ )
     cout<<*((int *)pvoid + i );

     // Same void pointer can be cast to char.
     cout<<"\n";
     pvoid = name;
     for( i=0; i < strlen( name); i++ )
          cout<< *((char *) pvoid + i));
    getch();
}
```

13

Chapter 13
String

Unit Structure

13.1 What is String?
13.2 The String Copy Function(strcpy())
13.3 The String Concatenation Function(strcat())
13.4 The String Compare Function(strcmp)

13.1 What is String?

A string is sequence of characters. Earlier version of C++ doesnot contain the built-in types for c++. The manipulation operation in strings is very tedious task in earlier version to this we have to define the array of character to string.

Now ANSCI standard provides a new class called string. For using class string, the program must include a header file knows as <string.h>

In c++, The string class is very large and include reach set of constructor, members function and operator.

The prototypes for three of string's most commonly used constructors are shown here:

```
string( );
string(const char *str);
string(const string &str);
```

The first constructor creates an empty string. The second constructor creates a string object from the null-terminated string pointed to by str.The third constructor creates a string from another string object.

Accessing String:

- We can also initialize a string by taking values from user using CIN command. But we must be aware about two things while initializing the string variable .
  - The First thing is the length of the string should not exceed the dimension or size of the character array. This is because

the C++ compiler doesn't check bounds on character arrays.
  - CIN is not capable of receiving multi-word string. Therefore string such as "hello world" would be unacceptable. This is because the CIN ">>" operator considers a space to be a terminating character. Thus it will read strings consisting of single word, but anything typed after a space is thrown away.

- We can overcome this problem of reading text containing spaces/ blanks in two ways.
  - We can use **gets()** and **puts()** functions instead of **cin>>** and **cout<<** respectively.
  - We can also use **cin.get()** function of the stream class.

Using gets(), puts() functions

```
void main()
     {
     char name[20];
     cout<<"enter  name of Person";
     gets(name);
     puts("hello");
     puts(name);
     cout<<name;
     }
```

Using cin.get() function

```
void main()
     {
     char name[20];
     cout<<"enter name";
     cin.get(name, 20);
     cout<<name;
     }
```

- There are a large set of useful string handling library functions provided by every C++ compiler. But we will only discuss four main functions. Which are
  - **strlen()** = It is use to finds the length of a string.
  - **strcpy()**= It is use to copies the string into another string .
  - **strcat()** = It is use to Append the string at the end of another string.
  - **strcmp()**= It compares two strings.

The String Length function(strlen())

- This function is used to counts the number of characters presents in a string.
- While calculating length of the string it doesn't count '\0' (null character).

```
void main()
     {
          char str1[ ]= "hello world";
          char str2[ ]= "India";
          int length1= strlen(str1);
          int length2= strlen(str2);
          cout<<length1;
          cout<<length2;
        }
```

13.2 THE STRING COPY FUNCTION(STRCPY())

- This function is used to copies the contents of one string into another string.

  strcpy(Destination , source)
- strcpy( ) goes on copying the each characters of source string into the destination string till it doesn't encounter '\0'. It is our responsibility to see that the destination string should be big enough to hold source string.

```
void main()
     {
          char str1[ ] = "India", str2[20];
          strcpy(str2,str1);
          cout<<str2;
     }
```

13.3 THE STRING CONCATENATION FUNCTION (STRCAT())

- The function is used to concatenate the source string at the end of destination string.

  strcat(destination, source);
- The destination string should be big enough to hold final string.

```
void main
     {
```

```
        char str1[ ] = "Hello";
        char str2[15 ] = "World";
        strcat(str2, str1);
        cout<<str2;
    }
```

13.4 THE STRING COMPARE FUNCTION(STRCMP)

- This function compares two string to find checks whether the two string are same or different.
- The two strings are compared character by character until there is a mismatch or end of one of the string is reached.
- If the two strings are equal it will return a value zero. If they are not than it returns the numeric difference between the ASCII value of non-matching characters.

Result	Condition
Less than zero	Str1<Str2
zero	Str1==Str2
Greater than zero	Str1>Str2

```
void main( )
        {
            char str1[ ]= "hello";
            char str2[ ]="world";
            int i=strcmp(str1, str2);
            cout<<i;
        }
```

14

Chapter 14
Basic Vectors

Unit Structure

14.1 Resizing Vectors
14.2 RESIZING VECTORS

14.1 RESIZING VECTORS

Arrays is use to store a group of values under a single name. The values can be any available data type (e.g., int, double, string, etc.). In C++, we talk about vectors, rather than arrays.

Vectors are declared with the following syntax:

vector<type> variable_name (num_of_elements);

The number of elements is optional. You could declare it like this:

vector<type> variable_name;

And that we would also declare an empty vector i.e a vector that contains zero elements.

The argument type in angle-brackets indicates the data type of the elements of the vector; variable_name is the name that we assign to the vector, and the optional num_of_elements may be provided to indicate how many elements the vector will initially contain.

Below are several examples of vector declarations:

```
vector<int> num (5);      // Declares a vector of 5 integers
vector<double> money (20);  // Declares a vector of 20 doubles
vector<string> names;       // Declares a vector of strings,
                            // initially empty (contains 0 strings)
```

When using vectors in our programs, we must provide the appropriate #include directive at the top of the file, since vectors are a Standard Library facility, and not a built-in part of the core language:

```
#include <vector.h>
```

After a vector has been declared specifying a certain number of elements, we can refer to individual elements in the vector using square brackets to provide a subscript or index, as shown below:

money[5]

When using a vector or array followed by square brackets with a subscript, the resulting expression refers to one individual element of the vector or array, as opposed to the group of values, you can use that expression as you would use a variable of the corresponding data type.

In the above example, the data type of the expression money[5] is double, so you can use it as you would use a variable of type double — you can assign a value to it (a numeric value, with or without decimals), or you can retrieve the value, use it for arithmetic operations, etc.

The above extends to other data types as well; if we have a vector of strings called names, the expression names[0] is a string, referring to the first element in the vector names. We can do anything with this expression that we would do with a string variable. For instance, the expression names[0].length() gives us the length of this string.

An important condition for the index or subscript is that it must indicate a valid element in the vector. Elements in a vector are "numbered" starting with element 0. This means that valid subscript values are numbers between 0 and size−1, where size is the number of elements of the vector. For the example above of grades, valid subscripts are between 0 and 19.

The following fragment shows an example of a program that asks the user for marks for a group of 20 students and stores them in a vector.

```
#include <iostream.h>
#include <vector.h>
int main()
{
   vector<double> students_roll(20);

   for (vector<double>::size_type i = 0; i < 20;
i++)
   {
        cout << "Enter roll number  of students
#" << i+1
          << ": " << flush;
        cin >> student_roll[i];
    }
   return 0;
}
```

The first statement declares a vector called student_rolls with capacity to hold 20 values of type double. These values can be accessed individually as students_roll[0] to students_roll[19]. The for loop has the counter i go from 0 to 19, allowing access to each individual element in a sequential manner, starting at 0 and going through each value from 0 to 19, inclusively.

Notice the data type for the subscript, vector<double>::size_type. As with strings, class vector<type> provides a size_type to represent positions and sizes. It is always recommended that you use this data type when dealing with vectors.

for loops usually go hand in hand with the use of vectors or arrays, as they provide a convenient way to access every element, one at a time, using the loop control variable as the subscript. This does not mean that we must use for loops whenever we require access to the elements of a vector — it only means that quite often, a for loop provides a convenient approach and we choose it as the mechanism to access the elements.

14.2 RESIZING VECTORS

Vectors have one important advantage with respect to C-style arrays: vectors can be resized during the execution of the program to accommodate any extra elements as needed, or even to “shrink” the vector.

In the example from the previous fragment above, if we don't know ahead of time (i.e., at the time we are writing the program) that there are 20 students, we could obtain that information at run-time (e.g., prompt the user for the number of students) and resize the vector accordingly, as shown below (though we notice that the example is somewhat silly, in that we could have waited until having the value of num_students and then declare the vector initializing it with that size):

```
vector<double> students_roll;
   // no size specified: vector contains
   // no elements

int num_students;
cout << "Number of students: " << flush;
cin >> num_students;

student_marks.resize (num_students);

for (vector<double>::size_type i = 0; i < num_students; i++)
{
   cout << "Enter marks for student #" << i+1
        << ": " << flush;
```

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

Notice that the valid subscripts for a vector with num_students elements are 0 to num_students−1. For that reason, the for loop starts at 0 and goes while i is less than num_elements.

It is always a better idea to control for loops using the size method of vector. That way, we make sure that we loop only through the right subscript values, and we avoid the risk of accidentally exceeding the limits of the vector:

```
for (vector<double>::size_type i = 0; i < student_marks.size(); i++)
```

The difference in this case seems insignificant, and it almost sounds unnecessary to use the size method; but again, it's always a good idea to stick to good programming practices that may be very convenient in larger or more complex programs.

In some situations, we can not determine the number of elements before reading them. That is, we may have to read numbers to then determine when to stop reading them (an example would be, keep reading values until you read a negative value). In such situations, the trick of resizing the vector is not an option (at least not the way it is used in the example above).

Vectors provide a convenient way of handling this type of situation. We can use the push_back method to append one element at the end of the array. The operation includes resizing to one more element to accommodate for the extra element, and storing the given value at the end of the array.

The example below shows the use of push_back to accept numbers from the user and store them in a vector, until the user indicates that there are no more numbers.

```
#include <iostream.h>
#include <vector.h>
int main()
{
    vector<double> student_marks;
    double mark;
    char answer;

    cout << "Enter marks (y/n)? " << flush;
    cin >> answer;

    while (answer == 'y' || answer == 'Y')
```

```
    {
        cout << "Enter value: " << flush;
        cin >> mark;

        student_marks.push_back (mark);

        cout << "More students (y/n)? " << flush;
        cin >> answer;
    }

    return 0;
}
```

15

Chapter 15
Structure

Unit Structure

15.1 Declaring The Structure
15.2 Combining Declaration and Definition
15.3 Initializing Structure Members
15.4 Structure As Function Arguments
15.5 Returning structure variables
15.6 Nested Structures
15.7 Array of Structures

- A structure is a collection of simple variables. The variables in a structure can be of different types: some can be int, some can be float and so on.
- The data items in a structure are called members of a structure.

15.1 DECLARING THE STRUCTURE

- The structure declaration tells how the structure is organized. It specifies what members the structure will have.

```
struct student
{
        char firstName[10];
        char lastName[10];
        float gpa;
        int regNumber;
};
```

Defining Structure Variable:

- We can define the structure variable same as variables by using built-in data type.

Example:

```
student  std1, std2;
```

- The above statement defines two variables std1 and std2 of type structure student.

- Once a structure variables has been defined, its members can be accessed using operator called DOT operator/ member access operator.

structVariable.memberName

Example:

```
std1.regNumber=536;
std1.firstName="Amit";      //error
std1.gpa=3.2;
std1.lastName="Tambe";  //error
```

Note: The first component of an expression involving the dot operator is the name of the specific structure variable not the name of the structure.

```
struct part
     {
          int modelNumber;
          int partNumber;
          float cost;
     };
     void main()
     {
     part  p1;
     p1.modelNumber=3215;
     p1.partNumber=10;
     p1.cost=1500;
            cout<<p1.modelNumber;
     cout<<p1.partNumber;
     cout<<p1.cost;
     getch();
}
```

15.2 COMBINING DECLARATION AND DEFINITION

- We can combine two statements of structure definition and declaration into a single statement.

```
struct
{
        int modelNumber;
        int partNumber;
        float cost;
}p1;
```

Note: you can remove the name of the structure in the declaration, if no more variables of this structure type will be defined later in the listing.

15.3 INITIALIZING STRUCTURE MEMBERS

```
struct part
     {
          char partName[10];
```

```
          int partNumber;
          float cost;
     };
void main()
{
     part p1={"abc", 25, 1500.0};
     cout<< p1.partName;
     cout<<p1.partNumber<<p1.cost;
}
```

Structure Variables in Assignment Statements:

- As normal variables, one structure variables can be assigned to another.

Example

part p1={"abc", 25, 1500.0}, p2;

p2=p1;

- In the above statement, the value of each member of p1 is assigned to the corresponding member of p2.

p2.cost= p1.cost;

- Similarly, you can also assign a single data member of one structure variable to the data member of other structure variable.

Note: one structure variable can be assigned to another only when both are of the same structure type.

15.4 STRUCTURE AS FUNCTION ARGUMENTS

```
struct part
{
     char partName[10];
     int partNumber;
     float cost;
};
void display(part);
void main()
{
     part p1;
     cin>>p1.partName;
     cin>>p1.partNumber;
            cin>>p1.cost;
     display(p1);
}
void display (part p2)
{
     cout<<p2.partName;
```

```
     cout<<p2.partNumber;
     cout<<p2.cost;
}
```

15.5 RETURNING STRUCTURE VARIABLES

- Like normal variables, you can also return the structure variables as a returning value of the function.

```
                part p1,p2;
     p1= display();   // function calling
     part display()   //function definition
     {
     part p3={"abc", 25, 1000.0};
     return p3;
     }
```

15.6 NESTED STRUCTURES

- One Structure can be nested into another structure.
- Example

```
struct Distance
     {
          int feet;
          float inches;
     };
     struct Room
     {
          char name[20];
          Distance length;
          Distance width;
     };
```

Accessing Nested Structure Members:

- To access the nested structure members, we have to apply dot operator twice.
-

Example

```
Room  Dining;
     strcpy(Dining.name,"dining room");
     Dining.length.feet=10;
     Dining.length.inches=6;
     Dining.width.feet=20;
```

Initializing Nested Structures

- In the previous example we can initialize the nested structure such that each structure of type **Distance,** which is embedded in **Room**, is initialized separately. Which involves surrounding the values with braces and separating them with commas.

- **Example;**
  Room dining ={"Dining room", {13, 6.5},{20,10} };

```
struct Distance
{
     int feet;
     float inches;
};
struct Room
{
     char name[20];
     Distance length;
     Distance width;
};
void main( )
{
     Room dining= {"Dining room",
                    {13,3.2},
                    {10,0.0} };
     cout<<dining.name<<endl;
     cout<<dining.length.feet<<endl;
     cout<< dining.length.inches<<endl;
     cout<< dining.width.feet<<endl;
     cout<<dining.width.inches<<endl;
}
```

15.7 ARRAY OF STRUCTURES

```
struct part
{
     int modelnumber;
     int partnumber;
     float cost;
};
void main ( )
{
     int n;
     part  abc[10];
for (n=0; n<10; n++)
     {
```

```
            cout<<   "Enter   Model   Number";
      cin>>abc[n].modelnumber;
            cout<<"Enter Part Number";
            cin>>abc[n].partnumber;
            cout<<"Enter Cost";
            cin>>abc[n].cost;
      }
}
```

15.8 PASSING STRING AS FUNCTION ARGUMENTS

```
void abc(char[ ]);
void main ()
{
      char str[20];
      cin>>str;
      abc(str);
{
void abc(char str1[ ])
{
      cout<<str1;
}
```
