UNIT 1 PROBLEM SOLVING

Structure

1.0 INTRODUCTION

In our daily life, we routinely encounter and solve problems. We pose problems that we need or want to solve. For this, we make use of available resources, and solve them. Some categories of resources include: the time and efforts of yours and others; tools; information; and money. Some of the problems that you encounter and solve are quite simple. But some others may be very complex.

In this unit we introduce you to the concepts of problem-solving, especially as they pertain to computer programming**.**

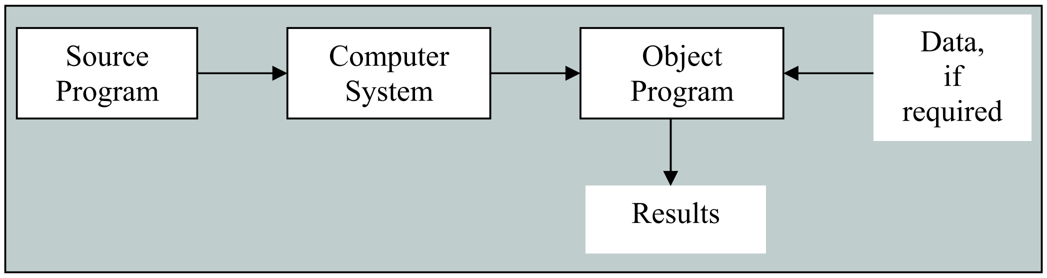
The problem-solving is a skill and there are no universal approaches one can take to solving problems. Basically one must explore possible avenues to a solution one by one until s/he comes across a right path to a solution. In general, as one gains experience in solving problems, one develops one's own techniques and strategies, though they are often intangible. Problem-solving skills are recognized as an integral component of computer programming. It is a demand and intricate process which is equally important throughout the project life cycle especially – study, designing, development, testing and implementation stages. The computer problem solving process requires:

- Problem anticipation
- Careful planning
- Proper thought process
- Logical precision
- Problem analysis
- Persistence and attention.

At the same time it requires personal creativity, analytic ability and expression. The chances of success are amplified when the problem solving is approached in a systematic way and satisfaction is achieved once the problem is satisfactorily solved. The problems should be anticipated in advance as far as possible and properly defined to help the algorithm definition and development process.

Computer is a very powerful tool for solving problems. It is a symbol-manipulating machine that follows a set of stored instructions called a program. It performs these manipulations very quickly and has memory for storing input, lists of commands and output. A computer cannot think in the way we associate with humans. When using the computer to solve a problem, you must specify the needed initial data, the operations which need to be performed (in order of performance) and what results you want for output. If any of these instructions are missing, you will get either no results or invalid results. In either case, your problem has not yet been solved. Therefore, several steps need to be considered before writing a program. These steps may free you from hours of finding and removing errors in your program (a process called **debugging**). It should also make the act of problem solving with a computer a much simpler task.

All types of computer programs are collectively referred to as **software**. Programming languages are also part of it. Physical computer equipment such as electronic circuitry, input/output devices, storage media etc. comes under **hardware**. Software governs the functioning of hardware. Operations performed by software may be built into the hardware, while instructions executed by the hardware may be generated in software. The decision to incorporate certain functions in the hardware and others in the software is made by the manufacturer and designer of the software and hardware. Normal considerations for this are: cost, speed, memory required, adaptability and reliability of the system. Set of instructions of the high level language used to code a problem to find its solution is referred to as **Source Program**. A translator program called **a compiler or interpreter**, translates the source program into the object program. This is the compilation or interpretation phase. All the testing of the source program as regards the correct format of instructions is performed at this stage and the errors, if any, are printed. If there is no error, the source program is transformed into the machine language program called **Object Program**. The Object Program is executed to perform calculations. This stage is the execution phase. Data, if required by the program, are supplied now and the results are obtained on the output device.



1.1 OBJECTIVES

After going through this unit, you should be able to:

- apply problem solving techniques;
- define an algorithm and its features;
- describe the analysis of algorithm efficiency;
- discuss the analysis of algorithm complexity; and
- design flowcharts.

1.2 PROBLEM - SOLVING TECHNIQUES

Problem solving is a creative process which defines systematization and mechanization. There are a number of steps that can be taken to raise the level of one's performance in problem solving.

1.2.1 Steps for Problem - Solving

A problem-solving technique follows certain steps in finding the solution to a problem. Let us look into the steps one by one:

Problem definition phase

The success in solving any problem is possible only after the problem has been fully understood. That is, we cannot hope to solve a problem, which we do not understand. So, the problem understanding is the first step towards the solution of the problem. In *problem definition phase*, we must emphasize *what must be done* rather than *how is it to be done*. That is, we try to extract the precisely defined set of tasks from the problem statement. Inexperienced problem solvers too often gallop ahead with the task of problem - solving only to find that they are either solving the wrong problem or solving just one particular problem.

Getting started on a problem

There are many ways of solving a problem and there may be several solutions. So, it is difficult to recognize immediately which path could be more productive. Sometimes you do not have any idea where to begin solving a problem, even if the problem has been defined. Such block sometimes occurs because you are overly concerned with the details of the implementation even before you have completely understood or worked out a solution. The best advice is not to get concerned with the details. Those can come later when the intricacies of the problem has been understood.

The use of specific examples

To get started on a problem, we can make use of heuristics i.e., the rule of thumb. This approach will allow us to start on the problem by picking a specific problem we wish to solve and try to work out the mechanism that will allow solving this particular problem. It is usually much easier to work out the details of a solution to a specific problem because the relationship between the mechanism and the problem is more clearly defined. This approach of focusing on a particular problem can give us the foothold we need for making a start on the solution to the general problem.

Similarities among problems

One way to make a start is by considering a specific example. Another approach is to bring the experience to bear on the current problem. So, it is important to see if there are any similarities between the current problem and the past problems which we have solved. The more experience one has the more tools and techniques one can bring to bear in tackling the given problem. But sometimes, it blocks us from discovering a desirable or better solution to the problem. A skill that is important to try to develop in problem - solving is the ability to view a problem from a variety of angles. One must be able to metaphorically turn a problem upside down, inside out, sideways, backwards, forwards and so on. Once one has developed this skill it should be possible to get started on any problem.

Working backwards from the solution

In some cases we can assume that we already have the solution to the problem and then try to work backwards to the starting point. Even a guess at the solution to the problem may be enough to give us a foothold to start on the problem. We can systematize the investigations and avoid duplicate efforts by writing down the various steps taken and explorations made. Another practice that helps to develop the problem solving skills is, once we have solved a problem, to consciously reflect back on the way we went about discovering the solution.

1.2.2 Using Computer as a Problem - Solving Tool

The computer is a resource - a versatile tool - that can help you solve some of the problems that you encounter. A computer is a very powerful general-purpose tool. Computers can solve or help to solve many types of problems. There are also many ways in which a computer can enhance the effectiveness of the time and effort that you are willing to devote to solving a problem. Thus, it will prove to be well worth the time and effort you spend to learn how to make effective use of this tool.

In this section, we discuss the steps involved in developing a program. Program development is a multi-step process that requires you to understand the problem, develop a solution, write the program, and then test it. This critical process determines the overall quality and success of your program. If you carefully design each program using good structured development techniques, your programs will be efficient, error-free, and easy to maintain. The following are the steps in detail:

1. Develop an *Algorithm* and a *Flowchart*.
2. Write the program in a computer language (for example say C programming language).
3. Enter the program using some editor.
4. Test and debug the program.
5. Run the program, input data, and get the results.

1.3 DESIGN OF ALGORITHMS

The first step in the program development is to devise and describe a precise plan of what you want the computer to do. This plan, expressed as a sequence of operations, is called an algorithm. An algorithm is just an outline or idea behind a program. something resembling C or Pascal, but with some statements in English rather than within the programming language. It is expected that one could translate each pseudo-code statement to a small number of lines of actual code, easily and mechanically.

1.3.1 Definition

An **algorithm** is a finite set of steps defining the solution of a particular problem. An algorithm is expressed in pseudocode - something resembling C language or Pascal, but with some statements in English rather than within the programming language. Developing an efficient algorithm requires lot of practice and skill. It must be noted that an efficient algorithm is one which is capable of giving the solution to the problem by using minimum resources of the system such as memory and processor's time. Algorithm is a language independent, well structured and detailed. It will enable the programmer to translate into a computer program using any high-level language.

1.3.2 Features of Algorithm

Following features should be present in an algorithm:

Proper understanding of the problem

For designing an efficient algorithm, the expectations from the algorithm should be clearly defined so that the person developing the algorithm can understand the expectations from it. This is normally the outcome of the problem definition phase.

Use of procedures / functions to emphasize modularity

To assist the development, implementation and readability of the program, it is usually helpful to modularize (section) the program. Independent functions perform specific and well defined tasks. In applying modularization, it is important to watch that the process is not taken so far to a point at which the implementation becomes difficult to read because of fragmentation. The program then can be implemented as calls to the various procedures that will be needed in the final implementations.

Choice of variable names

Proper variable names and constant names can make the program more meaningful and easier to understand. This practice tends to make the program more self documenting. A clear definition of all variables and constants at the start of the procedure / algorithm can also be helpful. For example, it is better to use variable *day* for the day of the weeks, instead of the variable *a* or something else.

Documentation of the program

Brief information about the segment of the code can be included in the program to facilitate debugging and providing information. A related part of the documentation is the information that the programmer presents to the user during the execution of the program. Since, the program is often to be used by persons who are unfamiliar with the working and input requirements of the program, proper documentation must be provided. That is, the program must specify what responses are required from the user. Care should also be taken to avoid ambiguities in these specifications. Also the program should "catch" incorrect responses to its requests and inform the user in an appropriate manner.

1.3.3 Criteria to be followed by an Algorithm

The following is the criteria to be followed by an algorithm:

- **Input:** There should be zero or more values which are to be supplied.
- **Output:** At least one result is to be produced.
- **Definiteness:** Each step must be clear and unambiguous.
- **Finiteness:** If we trace the steps of an algorithm, then for all cases, the algorithm must terminate after a finite number of steps.
- **Effectiveness:** Each step must be sufficiently basic that a person using only paper and pencil can in principle carry it out. In addition, not only each step is definite, it must also be feasible.

Example 1.1

Let us try to develop an algorithm to compute and display the sum of two numbers

1. Start
2. Read two numbers *a* and *b*
3. Calculate the sum of *a* and *b* and store it in *sum*
4. Display the value of *sum*
5. Stop

Example 1.2

Let us try to develop an algorithm to compute and print the average of a set of data values.

1. Start
2. Set the sum of the data values and the count to zero.

3. As long as the data values exist, add the next data value to the sum and add 1 to the count.
4. To compute the average, divide the sum by the count.
5. Display the average.
6. Stop

Example 1.3

Write an algorithm to calculate the factorial of a given number.

```
1. Start
2. Read the number n
3. [Initialize]
   i ← 1 , fact ← 1
4. Repeat steps 4 through 6 until i = n
5. fact ← fact * i
6. i ← i + 1
7. Print fact
8. Stop
```

Example 1.4

Write an algorithm to check that whether the given number is prime or not.

```
1. Start
2. Read the number num
3. [Initialize]
   i ← 2 , flag ← 1
4. Repeat steps 4 through 6 until i < num or flag = 0
5. rem ← num mod i
6. if rem = 0 then
   flag ← 0
      else
        i ← i + 1
7. if flag = 0 then
        Print Number is not prime
      Else
        Print Number is prime
8. Stop
```

1.3.4 Top Down Design

Once we have defined the problem and have an idea of how to solve it, we can then use the powerful techniques for designing algorithms. Most of the problems are complex or large problems and to solve them we have to focus on to comprehend at one time, a very limited span of logic or instructions. A technique for algorithm design that tries to accommodate this human limitation is known as **top-down design or stepwise refinement.**

Top down design provides the way of handling the logical complexity and detail encountered in computer algorithm. It allows building solutions to problems in step by step. In this way, specific and complex details of the implementation are encountered only at the stage when sufficient groundwork on the overall structure and relationships among the various parts of the problem.

Before the top down design can be applied to any problem, we must at least have the outlines of a solution. Sometimes this might demand a lengthy and creative investigation into the problem while at another time the problem description may in itself provide the necessary starting point for the top-down design.

Top-down design suggests taking the general statements about the solution one at a time, and then breaking them down into a more precise subtask / sub-problem. These sub-problems should more accurately describe how the final goal can be reached. The process of repeatedly breaking a task down into a subtask and then each subtask into smaller subtasks must continue until the sub-problem can be implemented as the program statement. With each spitting, it is essential to define how sub-problems interact with each other. In this way, the overall structure of the solution to the problem can be maintained. Preservation of the overall structure is important for making the algorithm comprehensible and also for making it possible to prove the correctness of the solution.

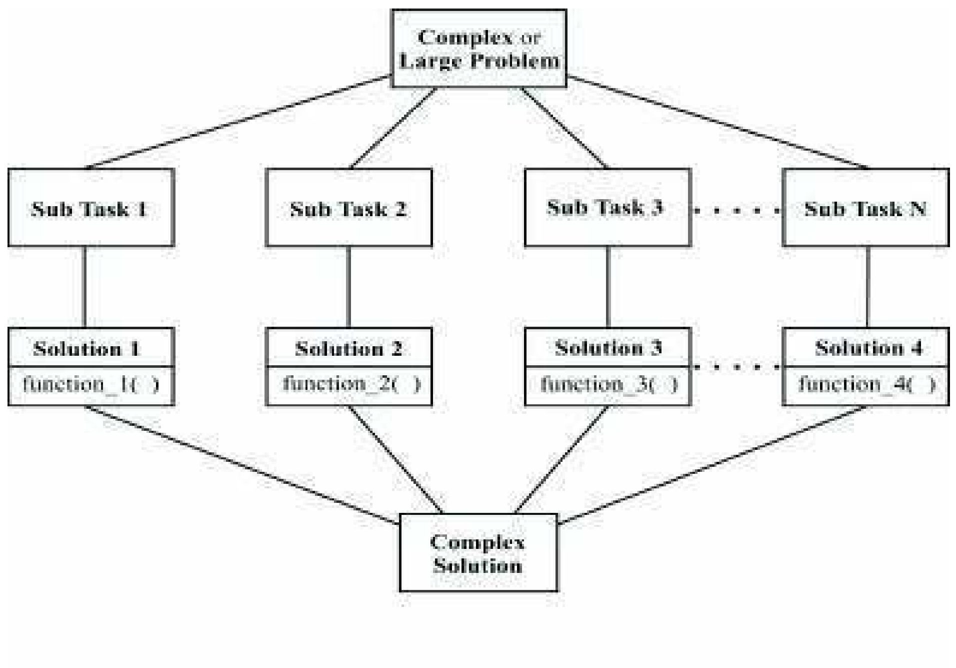


Figure 1.1: Schematic breakdown of a problem into subtasks as employed in top down design

1.4 ANALYSIS OF ALGORITHM EFFICENCY

Every algorithm uses some of the computer's resources like central processing time and internal memory to complete its task. Because of high cost of computing resources, it is desirable to design algorithms that are economical in the use of CPU time and memory. Efficiency considerations for algorithms are tied in with the design, implementation and analysis of algorithm. Analysis of algorithms is less obviously necessary, but has several purposes:

- Analysis can be more reliable than experimentation. If we experiment, we only know the behavior of a program on certain specific test cases, while analysis can give us guarantees about the performance on all inputs.
- It helps one choose among different solutions to problems. As we will see, there can be many different solutions to the same problem. A careful analysis and comparison can help us decide which one would be the best for our purpose, without requiring that all be implemented and tested.

- We can predict the performance of a program before we take the time to write code. In a large project, if we waited until after all the code was written to discover that something runs very slowly, it could be a major disaster, but if we do the analysis first we have time to discover speed problems and work around them.
- By analyzing an algorithm, we gain a better understanding of where the fast and slow parts are, and what to work on or work around in order to speed it up.

There is no simpler way of designing efficient algorithm, but a few suggestions as shown below can sometimes be useful in designing an efficient algorithm.

1.4.1 Redundant Computations

Redundant computations or unnecessary computations result in inefficiency in the implementation of the algorithms. When redundant calculations are embedded inside the loop for the variable which remains unchanged throughout the entire execution phase of the loop, the results are more serious. For example, consider the following code in which the value *a*a*a*c* is redundantly calculated in the loop:

```
x=0;
for i=0 to n
        x=x+1;
        y=(a*a*a*c)*x*x+b*b*x;
        print x,y
next i
```

This redundant calculation can be removed by small modification in the program:

```
x=0;
d=a*a*a*c;
e= b*b;
for i = 0 to n
        x = x+1;
        y = d*x*x+e*x;
        print x,y
next i
```

1.4.2 Referencing Array Elements

For using the array element, we require two memory references and an additional operation to locate the correct value for use. So, efficient program must not refer to the same array element again and again if the value of the array element does not change. We must store the value of array element in some variable and use that variable in place of referencing the array element. For example:

Version (1)

```
x=1;
for i = 0 to n
        if (a[i] > a[x]) x=i;
next i
max = a[x];
```

Version (2)

```
x=1;
max=a[1];
for i = 0 to n
        if(a[i]>max)
                x=i;
                max=a[i];
next i
```

Version (2) is more efficient algorithm than version (1) algorithm.

1.4.3 Inefficiency Due to Late Termination

Another place where inefficiency can come into an implementation is where considerably more tests are done than are required to solve the problem at hand. For example, if in the linear search process, all the list elements are checked for a particular element even if the point is reached where it was known that the element cannot occur later (in case of sorted list). Second example can be in case of the bubble sort algorithm, where the inner loop should not proceed beyond n-i, because last i elements are already sorted (in the algorithm given below).

```
for  i = 0 to n
        for j = 0 to n – 1
                if(a[j] > a[j+1])
                        //swap values a[j], a[j+1]
```

The efficient algorithm in which the inner loop terminates much before is given as:

```
for i=0 to n
        for j=0 to n – 1
                if(a[j]>a[j+1])
                        //swap values a[j], a[j+1]
```

1.4.4 Early Detection of Desired Output Condition

Sometimes the loops can be terminated early, if the desired output conditions are met. This saves a lot of unfruitful execution. For example, in the bubble sort algorithm, if during the current pass of the inner loop there are no exchanges in the data, then the list can be assumed to be sorted and the search can be terminated before running the outer loop for *n* times.

1.4.5 Trading Storage for Efficient Gains

A trade between storage and efficiency is often used to improve the performance of an algorithm. This can be done if we save some intermediary results and avoid having to do a lot of unnecessary testing and computation later on.

One strategy for speeding up the execution of an algorithm is to implement it using the least number of loops. It may make the program much harder to read and debug. It is therefore sometimes desirable that each loop does one job and sometimes it is required for computational speedup or efficiency that the same loop must be used for different jobs so as to reduce the number of loops in the algorithm. A kind of trade off is to be done while determining the approach for the same.

1.5 ANALYSIS OF ALGORITHM COMPLEXITY

Algorithms usually possess the following qualities and capabilities:

- Easily modifiable if necessary.
- They are easy, general and powerful.
- They are correct for clearly defined solution.
- Require less computer time, storage and peripherals i.e. they are more economical.
- They are documented well enough to be used by others who do not have a detailed knowledge of the inner working.
- They are not dependable on being run on a particular computer.
- The solution is pleasing and satisfying to its designer and user.
- They are able to be used as a sub-procedure for other problems.

Two or more algorithms can solve the same problem in different ways. So, quantitative measures are valuable in that they provide a way of comparing the performance of two or more algorithms that are intended to solve the same problem. This is an important step because the use of an algorithm that is more efficient in terms of time, resources required, can save time and money.

1.5.1 Computational Complexity

We can characterize an algorithm's performance in terms of the size (usually n) of the problem being solved. More computing resources are needed to solve larger problems in the same class. The table below illustrates the comparative cost of solving the problem for a range of n values.

$\log_2 n$	**n**	**$n \log_2 n$**	**n^2**	**n^3**	**2^n**
1	2	2	4	8	4
3.322	10	33.22	10^2	10^3	$>10^3$
6.644	10^2	664.4	10^4	10^6	$>>10^{25}$
9.966	10^3	9966.0	10^6	10^9	$>>10^{250}$
13.287	10^4	132877	10^8	10^{12}	$>>10^{2500}$

The above table shows that only very small problems can be solved with an algorithm that exhibit exponential behaviour. An exponential problem with n=100 would take immeasurably longer time. At the other extreme, for an algorithm with logarithmic dependency would merely take much less time (13 steps in case of $\log_2 n$ in the above table). These examples emphasize the importance of the way in which algorithms behave as a function of the problem size. Analysis of an algorithm also provides the theoretical model of the inherent computational complexity of a particular problem.

To decide how to characterize the behaviour of an algorithm as a function of size of the problem n, we must study the mechanism very carefully to decide just what constitutes the dominant mechanism. It may be the number of times a particular expression is evaluated, or the number of comparisons or exchanges that must be made as n grows. For example, comparisons, exchanges, and moves count most in sorting algorithm. The number of comparisons usually dominates so we use comparisons in computational model for sorting algorithms.

1.5.2 The Order of Notation

The O-notation gives an upper bound to a function within a constant factor. For a given function g(n), we denote by O(g(n)) the set of functions.

O(g(n)) = { f(n) : there exist positive constants c and n0, such that 0 <= f(n) <= cg(n) for all n >= n0 }

Using O-notation, we can often describe the running time of an algorithm merely by inspecting the algorithm's overall structure. For example a double nested loop structure of the following algorithm immediately yields O(n2) upper bound on the worst case running time.

```
for i=0 to n
        for j=0 to n
                print i,j
        next j
next i
```

What we mean by saying "the running time is $O(n^2)$" is that the worst case running time (which is a function of n) is $O(n^2)$. Or equivalently, no matter what particular input of size n is chosen for each value of n, the running time on that set of inputs is $O(n^2)$.

1.5.3 Rules for using the Big-O Notation

Big-O bounds, because they ignore constants, usually allow for very simple expression for the running time bounds. Below are some properties of big-O that allow bounds to be simplified. The most important property is that big-O gives an upper bound only. If an algorithm is $O(N^2)$, it doesn't have to take N^2 steps (or a constant multiple of N^2). But it can't take more than N^2. So any algorithm that is O(N), is also an $O(N^2)$ algorithm. If this seems confusing, think of big-O as being like "<". Any number that is < N is also $<N^2$.

1. Ignoring constant factors: O(c f(N)) = O(f(N)), where c is a constant; e.g. $O(20 N^3) = O(N^3)$
2. Ignoring smaller terms: If a<b then O(a+b) = O(b), for example, $O(N^2+N) = O(N^2)$
3. Upper bound only: If a<b then an O(a) algorithm is also an O(b) algorithm. For example, an O(N) algorithm is also an $O(N^2)$ algorithm (but not vice versa).
4. N and log N are bigger than any constant, from an asymptotic view (that means for large enough N). So if k is a constant, an O(N + k) algorithm is also O(N), by ignoring smaller terms. Similarly, an O(log N + k) algorithm is also O(log N).
5. Another consequence of the last item is that an O(N log N + N) algorithm, which is O(N(log N + 1)), can be simplified to O(N log N).

1.5.4 Worst and Average Case Behavior

Worst and average case behaviors of the algorithm are the two measures of performance that are usually considered. These two measures can be applied to both space and time complexity of an algorithm. The worst case complexity for a given problem of size n corresponds to the maximum complexity encountered among all problems of size n. For determination of the worst case complexity of an algorithm, we choose a set of input conditions that force the algorithm to make the least possible progress at each step towards its final goal.

In many practical applications it is very important to have a measure of the expected complexity of an algorithm rather than the worst case behavior. The expected complexity gives a measure of the behavior of the algorithm averaged over all possible problems of size n.

As a simple example: Suppose we wish to characterize the behavior of an algorithm that linearly searches an ordered list of elements for some value x.
1 2 3 4 5 … … …. N

In the worst case, the algorithm examines all n values in the list before terminating.

In the average case, the probability that x will be found at position 1 is 1/n, at position 2 is 2/n and so on. Therefore,

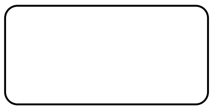
Average search cost = 1/n(1+2+3+ …..+n)
= 1/n(n/2(n+1)) = (n+1)/2

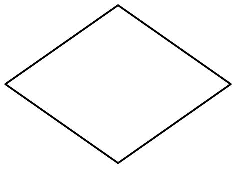
Let us see how to represent the algorithm in a graphical form using a flowchart in the following section.

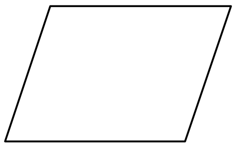
1.6 FLOWCHARTS

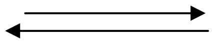
The next step after the algorithm development is the flowcharting. Flowcharts are used in programming to diagram the path in which information is processed through a computer to obtain the desired results. Flowchart is a graphical representation of an algorithm. It makes use of symbols which are connected among them to indicate the flow of information and processing. It will show the general outline of how to solve a problem or perform a task. It is prepared for better understanding of the algorithm.

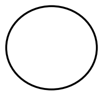
1.6.1 Basic Symbols used in flowchart design

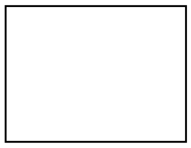

Start/Stop

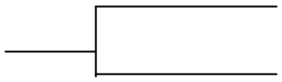

Question, Decision (Use in Branching)


Input/Output

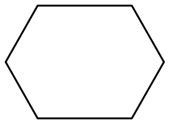

Lines or arrows represent the direction of the flow of control.

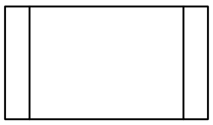

Connector (connect one part of the flowchart to another)


Process, Instruction


Comments, Explanations, Definitions.

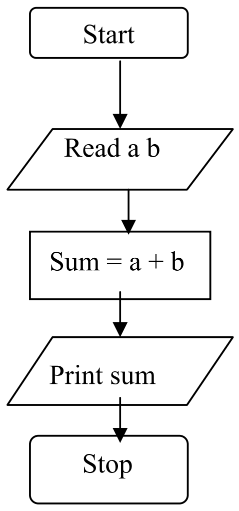
Additional Symbols Related to more advanced programming


Preparation (may be used with “do Loops”)


Refers to separate flowchart

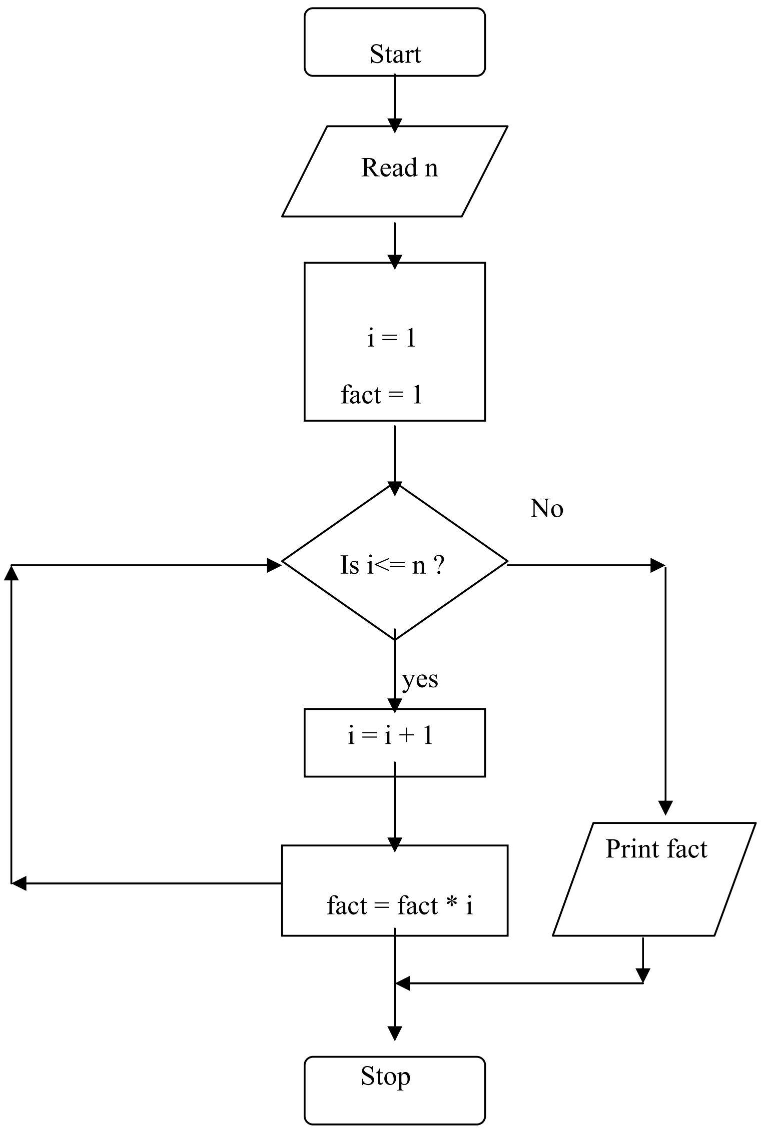
Example 1.5

The flowchart for the Example 1.1 is shown below:



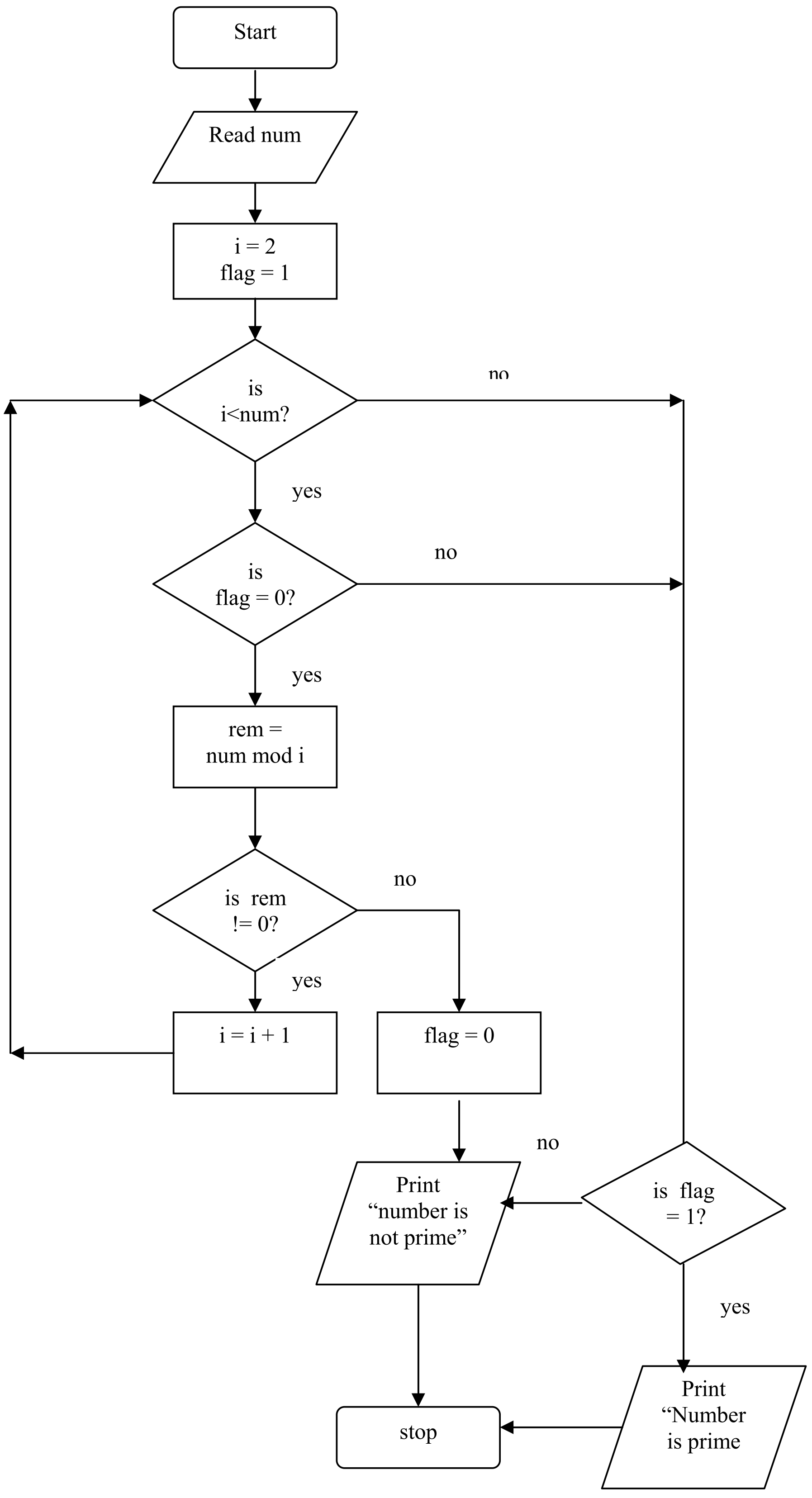
Example 1.6

The flowchart for the Example 1.3 (to find factorial of a given number) is shown below:



Example 1.7:

The flowchart for Example 1.4 is shown below:



Check Your Progress

1. Differentiate between flowchart and algorithm.

   ……………………………………………………………………………………

   ……………………………………………………………………………………

   ……………………………………………………………………………………

   ……………………………………………………………………………………

2. Compute and print the sum of a set of data values.

   ……………………………………………………………………………………

   ……………………………………………………………………………………

   ……………………………………………………………………………………

   ……………………………………………………………………………………

3. Write the following steps are suggested to facilitate the problem solving process using computer.

   ……………………………………………………………………………………

   ……………………………………………………………………………………

   ……………………………………………………………………………………

   ……………………………………………………………………………………

4. Draw an algorithm and flowchart to calculate the roots of quadratic equation Ax^2 + Bx + C = 0.

   ……………………………………………………………………………….…

   …………………………………………………………………………………….

   ……………………………………………………………………………………

   ……………………………………………………………………………………

1.7 SUMMARY

To solve a problem different problem - solving tools are available that help in finding the solution to problem in an efficient and systematic way. Steps should be followed to solve the problem that includes writing the algorithm and drawing the flowchart for the solution to the stated problem**.** Top down design provides the way of handling the logical complexity and detail encountered in computer algorithm. It allows building solutions to problems in a stepwise fashion. In this way, specific and complex details of the implementation are encountered only at the stage when sufficient groundwork on the overall structure and relationships among the carious parts of the problem. We present C language - a standardized, industrial-strength programming language known for its power and portability as an implementation vehicle for these problem solving techniques using computer.

1.8 SOLUTIONS / ANSWERS

Check Your Progress

1. The process to devise and describe a precise plan (in the form of sequence of operations) of what you want the computer to do, is called an **algorithm**. An algorithm may be symbolized in a flowchart or pseudocode.

2. 1. Start
   2. Set the sum of the data values and the count of the data values to zero.
   3. As long as the data values exist, add the next data value to the sum and add 1 to the count.
   4. Display the average.
   5. Stop

3. The following steps are suggested to facilitate the problem solving process:

   a) Define the problem
   b) Formulate a mathematical model
   c) Develop an algorithm
   d) Design the flowchart
   e) Code the same using some computer language
   f) Test the program

1.9 FURTHER READINGS

1. How to solve it by Computer, 5th Edition, *R G Dromey*, PHI, 1992.
2. Introduction to Computer Algorithms, Second Edition, *Thomas H. Cormen*, MIT press, 2001.
3. Fundamentals of Algorithmics, *Gilles Brassword, Paul Bratley,* PHI, 1996.
4. Fundamental Algorithms, Third Edition, *Donald E Knuth*, Addison-Wesley, 1997.

UNIT 2 BASICS OF C

Structure

2.0 INTRODUCTION

In the earlier unit we introduced you to the concepts of problem-solving, especially as they pertain to computer programming**.** In this unit we present C language - a standardized, industrial-strength programming language known for its power and portability as an implementation vehicle for these problem solving techniques using computer.

A language is a mode of communication between two people. It is necessary for those two people to understand the language in order to communicate. But even if the two people do not understand the same language, a translator can help to convert one language to the other, understood by the second person. Similar to a translator is the mode of communication between a user and a computer is a computer language. One form of the computer language is understood by the user, while in the other form it is understood by the computer. A translator (or compiler) is needed to convert from user's form to computer's form. Like other languages, a computer language also follows a particular grammar known as the syntax.

In this unit we will introduce you the basics of programming language C.

2.1 OBJECTIVES

After going through this unit you will be able to:

- define what is a program?
- understand what is a C programming language?
- compile a C program;
- identify the syntax errors;
- run a C program; and
- understand what are run time and logical errors.

2.2 WHAT IS A PROGRAM AND WHAT IS A PROGRAMMING LANGUAGE?

We have seen in the previous unit that a computer has to be fed with a detailed set of instructions and data for solving a problem. Such a procedure which we call an *algorithm* is a series of steps arranged in a logical sequence. Also we have seen that a *flowchart* is a pictorial representation of a sequence of instructions given to the computer. It also serves as a document explaining the procedure used to solve a problem. In practice it is necessary to express an algorithm using a *programming language*. A procedure expressed in a programming language is known as a *computer program*.

Computer programming languages are developed with the primary objective of facilitating a large number of people to use computers without the need for them to know in detail the internal structure of the computer. Languages are designed to be *machine-independent*. Most of the programming languages ideally designed, to execute a program on any computer regardless of who manufactured it or what model it is.

Programming languages can be divided into two categories:

(i) **Low Level Languages or Machine Oriented Languages:** The language whose design is governed by the circuitry and the structure of the machine is known as the **Machine language**. This language is difficult to learn and use. It is specific to a given computer and is different for different computers i.e. these languages are **machine-dependent**. These languages have been designed to give a better machine efficiency, i.e. faster program execution. Such languages are also known as Low Level Languages. Another type of Low-Level Language is the Assembly Language. We will code the assembly language program in the form of mnemonics. Every machine provides a different set of mnemonics to be used for that machine only depending upon the processor that the machine is using.

(ii) **High Level Languages or Problem Oriented Languages:** These languages are particularly oriented towards describing the procedures for solving the problem in a concise, precise and unambiguous manner. Every high level language follows a precise set of rules. They are developed to allow application programs to be run on a variety of computers. These languages are *machine-independent*. Languages falling in this category are FORTRAN, BASIC, PASCAL etc. They are easy to learn and programs may be written in these languages with much less effort. However, the computer cannot understand them and they need to be translated into machine language with the help of other programs known as Compilers or Translators.

2.3 C LANGUAGE

Prior to writing C programs, it would be interesting to find out what really is C language, how it came into existence and where does it stand with respect to other computer languages. We will briefly outline these issues in the following section.

2.3.1 History of C

C is a programming language developed at AT&T's Bell Laboratory of USA in 1972. It was designed and written by Dennis Ritchie. As compared to other programming languages such as Pascal, C allows a precise control of input and output.

Now let us see its historical development. The late 1960s were a turbulent era for computer systems research at Bell Telephone Laboratories. By 1960, many programming languages came into existence, almost each for a specific purpose. For example COBOL was being used for Commercial or Business Applications, FORTRAN for Scientific Applications and so on. So, people started thinking why could not there be a one general purpose language. Therefore, an International Committee was set up to develop such a language, which came out with the invention of ALGOL60. But this language never became popular because it was too abstract and too general. To improve this, a new language called Combined Programming Language (CPL) was developed at Cambridge University. But this language was very complex in the sense that it had too many features and it was very difficult to learn. Martin Richards at Cambridge University reduced the features of CPL and developed a new language called Basic Combined Programming Language (BCPL). But unfortunately it turned out to be much less powerful and too specific. Ken Thompson at AT & T's Bell Labs, developed a language called B at the same time as a further simplification of CPL. But like BCPL this was also too specific. Ritchie inherited the features of B and BCPL and added some features on his own and developed a language called C. C proved to be quite compact and coherent. Ritchie first implemented C on a DEC PDP-11 that used the UNIX Operating System.

For many years the *de facto* standard for C was the version supplied with the UNIX version 5 operating system. The growing popularity of microcomputers led to the creation of large number of C implementations. At the source code level most of these implementations were highly compatible. However, since no standard existed there were discrepancies. To overcome this situation, ANSI established a committee in 1983 that defined an ANSI standard for the C language.

2.3.2 Salient features of C

C is a general purpose, structured programming language. Among the two types of programming languages discussed earlier, C lies in between these two categories. That's why it is often called a ***middle level language.*** It means that it combines the elements of high level languages with the functionality of assembly language. It provides relatively good programming efficiency (as compared to machine oriented language) and relatively good machine efficiency as compared to high level languages). As a middle level language, C allows the manipulation of bits, bytes and addresses – the basic elements with which the computer executes the inbuilt and memory management functions. C code is very portable, that it allows the same C program to be run on machines with different hardware configurations. The flexibility of C allows it to be used for systems programming as well as for application programming.

C is commonly called a structured language because of structural similarities to ALGOL and Pascal. The distinguishing feature of a structured language is compartmentalization of code and data. Structured language is one that divides the entire program into modules using top-down approach where each module executes one job or task. It is easy for debugging, testing, and maintenance if a language is a structured one. C supports several control structures such as **while, do-while and for** and various data structures such as **strucs, files, arrays** etc. as would be seen in the later units. The basic unit of a C program is a **function -** C's standalone subroutine**.** The structural component of C makes the programming and maintenance easier.

Check Your Progress 1

1. "A Program written in Low Level Language is faster." Why?
   ………………………………………………………………………………………
   ………………………………………………………………………………………
   ………………………………………………………………………………………

2. What is the difference between high level language and low level language?
………………………………………………………………………………………
………………………………………………………………………………………
………………………………………………………………………………………

3. Why is C referred to as middle level language?
………………………………………………………………………………………
………………………………………………………………………………………
………………………………………………………………………………………

2.4 STRUCTURE OF A C PROGRAM

As we have already seen, to solve a problem there are three main things to be considered. Firstly, what should be the output? Secondly, what should be the inputs that will be required to produce this output and thirdly, the steps of instructions which use these inputs to produce the required output. As stated earlier, every programming language follows a set of rules; therefore, a program written in C also follows predefined rules known as syntax. C is a case sensitive language**.** All C programs consist of one or more functions. One function that must be present in every C program is **main()**. This is the first function called up when the program execution begins. Basically, **main()** outlines what a program does. Although **main** is not given in the keyword list,it cannot be used for naming a variable. The structure of a C program is illustrated in Figure.2.1 where functions func1( ) through funcn( ) represent user defined functions.

```
 Preprocessor directives
Global data declarations
 main ( )      /* main function*/
 {
         Declaration part;

         Program statements;
  }

 /*User defined functions*/
 func1( )
 {
         …………
 }

 func2 ( )
 {
         …………
  }
  .
  .
  .
   funcn ( )
   {
         …………
   }
```

Figure. 2.1: Structure of a C Program.

A Simple C Program

From the above sections, you have become familiar with, a programming language and structure of a C program. It's now time to write a simple C program. This program will illustrate how to print out the message "This is a C program".

Example 2.1: Write a program to print a message on the screen.

```
/*Program to print a message*/
#include <stdio.h>                /* header file*/
main()                            /* main function*/
{
  printf("This is a C program\n");  /* output statement*/
}
```

Though the program is very simple, a few points must be noted.

Every C program contains a function called **main()**. This is the starting point of the program. This is the point from where the execution begins. It will usually call other functions to help perform its job, some that we write and others from the standard libraries provided.

#include <stdio.h> is a reference to a special file called stdio.h which contains information that must be included in the program when it is compiled. The inclusion of this required information will be handled automatically by the compiler. You will find it at the beginning of almost every C program. Basically, all the statements starting with # in a C program are called preprocessor directives. These will be considered in the later units. Just remember, that this statement allows you to use some predefined functions such as, *printf(),* in this case.

main() declares the start of the function, while the two curly brackets { } shows the start and finish of the function. Curly brackets in C are used to group statements together as a function, or in the body of a loop. Such a grouping is known as a compound statement or a block. Every statement within a function ends with a terminator semicolon (;).

printf("This is a C program\n"); prints the words on the screen. The text to be printed is enclosed in double quotes. The **\n** at the end of the text tells the program to print a newline as part of the output. That means now if we give a second printf statement, it will be printed in the next line.

Comments may appear anywhere within a program, as long as they are placed within the delimiters **/*** and ***/.** Such comments are helpful in identifying the program's principal features or in explaining the underlying logic of various program features. While useful for teaching, such a simple program has few practical uses. Let us consider something rather more practical. Let us look into the example given below, the complete program development life cycle.

Example 2.1

Develop an algorithm, flowchart and program to add two numbers.

Algorithm

1. Start
2. Input the two numbers ***a*** and ***b***
3. Calculate the sum as ***a+b***
4. Store the result in ***sum***

5. Display the result
6. Stop.

Flowchart

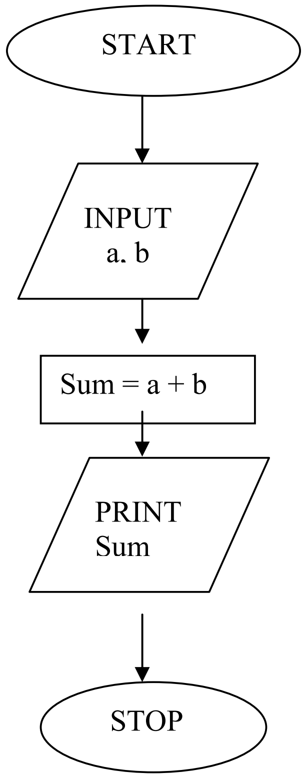


Figure 2.2: Flow chart to add two numbers

Program

```
#include <stdio.h>

main()
{
  int a,b,sum;                    /* variables declaration*/

  printf("\n Enter the values for a and b: \n");
  scanf("%d, %d", &a, &b);

  sum=a+b;

  printf("\nThe sum is %d",sum);    /*output statement*/
}
```

OUTPUT

```
Enter the values of a and b:
2 3
The sum is  5
```

In the above program considers two variables *a* and *b*. These variables are declared as integers **(int)**, it is the data type to indicate integer values. Next statement is the printf statement meant for prompting the user to input the values of ***a*** and ***b***. scanf is the function to intake the values into the program provided by the user. Next comes the processing / computing part which computes the **sum**. Again the **printf** statement is a bit different from the first program; it includes a format specifier (%d). The format specifier indicates the kind of value to be printed. We will study about other data types and format specifiers in detail in the following units. In the printf statement above, sum is not printed in double quotes because we want its value to be printed. The number of format specifiers and the variable should match in the printf statement.

At this stage, don't go much in detail. However, in the following units you will be learning all these details.

2.5 WRITING A C PROGRAM

A C program can be executed on platforms such as DOS, UNIX etc. DOS stores C program with a file extension ***.c***. Program text can be entered using any text editor such as EDIT or any other. To edit a file called ***testprog.c*** using edit editor, gives:

C:> edit   testprog.c

If you are using **Turbo C**, then Turbo C provides its own editor which can be used for writing the program. Just give the full pathname of the executable file of Turbo C and you will get the editor in front of you. For example:

C:> turboc\bin\tc

Here, tc.exe is stored in bin subdirectory of turboc directory. After you get the menu just type the program and store it in a file using the menu provided. The file automatically gets the extension of .c.

UNIX also stores C program in a file with extension is ***.c***. This identifies it as a C program. The easiest way to enter your text is using a text editor like *vi*, *emacs* or *xedit*. To edit a file called testprog.c using vi type

$ vi   testprog.c

The editor is also used to make subsequent changes to the program.

2.6 COMPILING A C PROGRAM

After you have written the program the next step is to save the program in a file with extension . c . This program is in high-level language. But this language is not understood by the computer. So, the next step is to convert the high-level language program (source code) to machine language (object code). This task is performed by a software or program known as a **compiler**. Every language has its own compiler that converts the source code to object code. The compiler will compile the program successfully if the program is syntactically correct; else the object code will not be produced. This is explained pictorially in Figure 2.3.

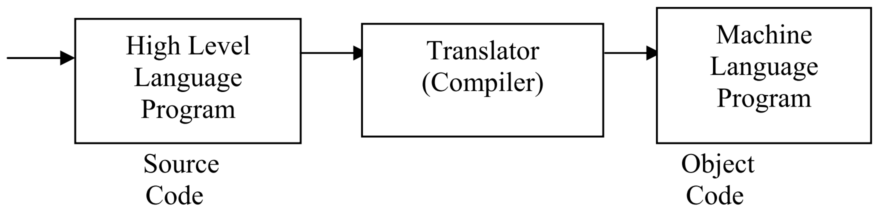


Figure 2.3: Process of Translation

2.6.1 The C Compiler

If you are working on UNIX platform, then if the name of the program file is **testprog.c**, to compile it, the simplest method is to type

```
cc testprog.c
```

This will compile testprog.c, and, if successful, will produce a executable file called ***a.out***. If you want to give the executable file any other, you can type

```
cc testprog.c -o testprog
```

This will compile ***testprog.c***, creating an executable file testprog.

If you are working with TurboC on DOS platform then the option for compilation is provided on the menu. If the program is syntactically correct then this will produce a file named as **testprog.obj**. If not, then the syntax errors will be displayed on the screen and the object file will not be produced. The errors need to be removed before compiling the program again. This process of removing the errors from the program is called as the **debugging**.

2.6.2 Syntax and Semantic Errors

Every language has an associated grammar, and the program written in that language has to follow the rules of that grammar. For example in English a sentence such a "Shyam, is playing, with a ball". This sentence is syntactically incorrect because commas should not come the way they are in the sentence.

Likewise, C also follows certain syntax rules. When a C program is compiled, the compiler will check that the program is syntactically correct. If there are any syntax errors in the program, those will be displayed on the screen with the corresponding line numbers.

Let us consider the following program.

Example 2.3: Write a program to print a message on the screen.

```
/* Program to print a message on the screen*/

#include <stdio.h

main( )
{
  printf("Hello, how are you\n")
```

Let the name of the program be **test.c** .If we compile the above program as it is we will get the following errors:

```
Error test.c 1:No file name ending
Error test.c 5: Statement missing ;
Error test.c 6: Compound statement missing }
```

Edit the program again, correct the errors mentioned and the corrected version appears as follows:

```
#include <stdio.h>
main( )
{
  printf ("Hello, how are you\n");
}
```

Apart from syntax errors, another type of errors that are shown while compilation are semantic errors. These errors are displayed as warnings. These errors are shown if a particular statement has no meaning. The program does compile with these errors, but it is always advised to correct them also, since they may create problems while execution. The example of such an error is that say you have declared a variable but have not used it, and then you get a warning "code has no effect". These variables are unnecessarily occupying the memory.

Check Your Progress 2

1. What is the basic unit of a C program?
   ……………………………………………………………………………………
   ……………………………………………………………………………………
   ……………………………………………………………………………………
2. "The program is syntactically correct". What does it mean?
   ……………………………………………………………………………………
   ……………………………………………………………………………………
   ……………………………………………………………………………………
3. Indicate the syntax errors in the following program code:

```
include <stdio.h>

main( )
[
  printf("hello\n");
]
```

   ……………………………………………………………………………………
   ……………………………………………………………………………………
   ……………………………………………………………………………………

2.7 LINK AND RUN THE C PROGRAM

After compilation, the next step is linking the program. Compilation produces a file with an extension **.obj**. Now this **.obj** file cannot be executed since it contains calls to functions defined in the standard library (header files) of C language. These functions have to be linked with the code you wrote. C comes with a standard library that provides functions that perform most commonly needed tasks. When you call a function that is not the part of the program you wrote, C remembers its name. Later the linker combines the code you wrote with the object code already found in the standard library. This process is called *linking.* In other words, Linker is a program that links separately compiled functions together into one program. It combines the functions in the standard C library with the code that you wrote. The output of the linker in an executable program i.e., a file with an extension ***.exe***.

2.7.1 Run the C Program Through the Menu

When we are working with TurboC in DOS environment, the menu in the GUI that pops up when we execute the executable file of TurboC contains several options for executing the program:

i) Link , after compiling
ii) Make, compiles as well as links
iii) Run

All these options create an executable file and when these options are used we also get the output on user screen. To see the output we have to shift to user screen window.

2.7.2 Run From an Executable File

An ***.exe*** file produced by can be directly executed.

UNIX also includes a very useful program called **make**. **Make** allows very complicated programs to be compiled quickly, by reference to a configuration file (usually called makefile). If your C program is a single file, you can usually use make by simply typing –

```
make testprog
```

This will compile **testprog.c** as well as link your program with the standard library so that you can use the standard library functions such as printf and put the executable code in **testprog.**

In case of DOS environment , the options provided above produce an executable file and this file can be directly executed from the DOS prompt just by typing its name without the extension. That is if the name of the program is test.c, after compiling and linking the new file produced is test.exe only if compilation and linking is successful.

This can be executed as:

```
c>test
```

2.7.3 Linker Errors

If a program contains syntax errors then the program does not compile, but it may happen that the program compiles successfully but we are unable to get the executable file, this happens when there are certain linker errors in the program. For example, the object code of certain standard library function is not present in the standard C library; the definition for this function is present in the header file that is why we do not get a compiler error. Such kinds of errors are called linker errors. The executable file would be created successfully only if these linker errors are corrected.

2.7.4 Logical and Runtime Errors

After the program is compiled and linked successfully we execute the program. Now there are three possibilities:

1) The program executes and we get correct results,
2) The program executes and we get wrong results, and
3) The program does not execute completely and aborts in between.

The first case simply means that the program is correct. In the second case, we get wrong results; it means that there is some logical mistake in our program. This kind of error is known as **logical error**. This error is the most difficult to correct. This error is corrected by debugging. Debugging is the process of removing the errors from the program. This means manually checking the program step by step and verifying the results at each step. Debugging can be made easier by a tracer provided in Turbo C environment. Suppose we have to find the average of three numbers and we write the following code:

Example 2.4: Write a C program to compute the average of three numbers

```
/* Program to compute average of three numbers *?
#include<stdio.h>
```

```
main( )
 {
   int a,b,c,sum,avg;

   a=10;
   b=5;
   c=20;

   sum = a+b+c;
   avg = sum / 3;
   printf("The average is %d\n", avg);
}
```

OUTPUT

```
The average is 8.
```

The exact value of average is 8.33 and the output we got is 8. So we are not getting the actual result, but a rounded off result. This is due to the logical error. We have declared variable **avg** as an integer but the average calculated is a real number, therefore only the integer part is stored in **avg.** Such kinds of errors which are not detected by the compiler or the linker are known as **logical errors**.

The third kind of error is only detected during execution. Such errors are known as **run time errors**. These errors do not produce the result at all, the program execution stops in between and the run time error message is flashed on the screen. Let us look at the following example:

Example 2.5: **Write a program to divide a sum of two numbers by their difference**

```
/* Program to divide a sum of two numbers by their difference*/

#include <stdio.h>

 main( )
 {

   int a,b;
   float c;

   a=10;
   b=10;

   c = (a+b) / (a-b);
   printf("The value of the result is %f\n",c);
}
```

The above program will compile and link successfully, it will execute till the first *printf* statement and we will get the message in this statement, as soon as the next statement is executed we get a runtime error of "Divide by zero" and the program halts. Such kinds of errors are **runtime errors**.

2.8 DIAGRAMMATIC REPRESENTATION OF PROGRAM EXECUTION PROCESS

The following figure 2.4 shows the diagrammatic representation of the program execution process.

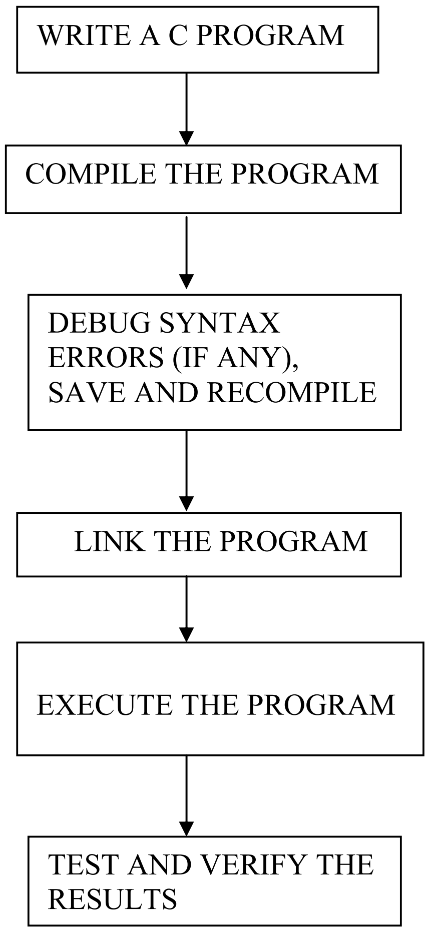

WRITE A C PROGRAM
COMPILE THE PROGRAM
DEBUG SYNTAX ERRORS (IF ANY), SAVE AND RECOMPILE
LINK THE PROGRAM
EXECUTE THE PROGRAM
TEST AND VERIFY THE RESULTS

Figure 2.4: Program Execution Process

Check Your Progress 3

1. What is the extension of an executable file?
   ……………………………………………………………………………………
   ……………………………………………………………………………………
   ……………………………………………………………………………………
2. What is the need for linking a compiled file?
   ……………………………………………………………………………………
   ……………………………………………………………………………………
   ……………………………………………………………………………………
3. How do you correct the logical errors in the program?
   ……………………………………………………………………………………
   ……………………………………………………………………………………
   ……………………………………………………………………………………

2.9 SUMMARY

In this unit, you have learnt about a program and a programming language. You can now differentiate between high level and low level languages. You can now define what is C, features of C. You have studied the emergence of C. You have seen how C is different, being a middle level Language, than other High Level languages. The advantage of high level language over low level language is discussed.

You have seen how you can convert an algorithm and flowchart into a C program. We have discussed the process of writing and storing a C program in a file in case of UNIX as well as DOS environment.

You have learnt about compiling and running a C program in UNIX as well as on DOS environment. We have also discussed about the different types of errors that are encountered during the whole process, i.e. syntax errors, semantic errors, logical errors, linker errors and runtime errors. You have also learnt how to remove these errors. You can now write simple C programs involving simple arithmetic operators and the *printf( )* statement. With these basics, now we are ready to learn the C language in detail in the following units.

2.10 SOLUTIONS / ANSWERS

Check Your Progress 1

1. A program written in Low Level Language is faster to execute since it needs no conversion while a high level language program need to be converted into low level language.

2. Low level languages express algorithms on the form of numeric or mnemonic codes while High Level Languages express algorithms in the using concise, precise and unambiguous notation. Low level languages are machine dependent while High level languages are machine independent. Low level languages are difficult to program and to learn, while High level languages are easy to program and learn. Examples of High level languages are FORTRAN, Pascal and examples of Low level languages are machine language and assembly language.

3. C is referred to as middle level language as with C we are able to manipulate bits, bytes and addresses i.e. interact with the hardware directly. We are also able to carry out memory management functions.

Check Your Progress 2

1. The basic unit of a C program is a C function.

2. It means that program contains no grammatical or syntax errors.

3. Syntax errors:

   a) # not present with include
   b) {brackets should be present instead of [brackets.

Check Your Progress 3

1. The extension of an executable file is .exe.

2. The C program contains many C pre-defined functions present in the C library. These functions need to be linked with the C program for execution; else the C program may give a linker error indicating that the function is not present.

3. Logical errors can be corrected through debugging or self checking.

2.11 FURTHER READINGS

1. The C Programming Language, *Kernighan & Richie*, PHI Publication.
2. Programming with C, Second Edition, *Byron Gottfried*, Tata Mc Graw Hill, 2003.
3. The C Complete Reference, Fourth Editon, *Herbert Schildt*, Tata Mc Graw Hill, 2002.
4. Programming with ANSI and Turbo C, *Ashok N. Kamthane*, Pearson Education Asia, 2002.
5. Computer Science A structured programming approach using C Second Edition, *Behrouza A. Forouzan, Richard F. Gilberg*, Brooks/Cole, Thomson Learning, 2001.

UNIT 3 VARIABLES AND CONSTANTS

Structure

3.0 INTRODUCTION

As every natural language has a basic character set, computer languages also have a character set, rules to define words. Words are used to form statements. These in turn are used to write the programs.

Computer programs usually work with different types of data and need a way to store the values being used. These values can be numbers or characters. C language has two ways of storing number values—**variables and constants**—with many options for each. Constants and variables are the fundamental elements of each program. Simply speaking, a program is nothing else than defining them and manipulating them. A variable is a data storage location that has a value that can change during program execution. In contrast, a constant has a fixed value that can't change.

This unit is concerned with the basic elements used to construct simple C program statements. These elements include the C character set, identifiers and keywords, data types, constants, variables and arrays, declaration and naming conventions of variables.

3.1 OBJECTIVES

After going through this unit, you will be able to:

- define identifiers, data types and keywords in C;
- know name the identifiers as per the conventions;
- describe memory requirements for different types of variables; and
- define constants, symbolic constants and their use in programs.

3.2 CHARACTER SET

When you write a program, you express C source files as text lines containing characters from the character set. When a program executes in the target environment, it uses characters from the character set. These character sets are related, but need not have the same encoding or all the same members.

Every character set contains a distinct code value for each character in the **basic C character set**. A character set can also contain additional characters with other code values. The C language character set has alphabets, numbers, and special characters as shown below:

1. Alphabets including both lowercase and uppercase alphabets - A-Z and a-z.

2. Numbers 0-9

3. Special characters include:

;	:	{	,	'	"	|
}	>	<	/	\	~	_
[	]	!	$	?	*	+
=	(	)	-	%	#	^
@	&	.				

3.3 IDENTIFIERS AND KEYWORDS

Identifiers are the names given to various program elements such as constants, variables, function names and arrays etc. Every element in the program has its own distinct name but one cannot select any name unless it conforms to valid name in C language. Let us study first the rules to define names or identifiers.

3.3.1 Rules for Forming Identifiers

Identifiers are defined according to the following rules:

1. It consists of letters and digits.
2. First character must be an alphabet or underscore.
3. Both upper and lower cases are allowed. Same text of different case is not equivalent, for example: **TEXT** is not same as **text**.
4. Except the special character underscore (_), no other special symbols can be used.

For example, some valid identifiers are shown below:

X
X123
_XI
temp
tax_rate

For example, some invalid identifiers are shown below:

123	First character to be alphabet.
"X."	Not allowed.
order-no	Hyphen allowed.
error flag	Blankspace allowed.

3.3.2 Keywords

Keywords are reserved words which have standard, predefined meaning in C. They cannot be used as program-defined identifiers.

The lists of C keywords are as follows:

char	while	do	typedef	auto
int	if	else	switch	case
printf	double	struct	break	static
long	enum	register	extern	return
union	const	float	short	unsigned
continue	for	signed	void	default
goto	sizeof	volatile		

Note: Generally all keywords are in lower case although uppercase of same names can be used as identifiers.

3.4 DATA TYPES AND STORAGE

To store data inside the computer we need to first identify the type of data elements we need in our program. There are several different types of data, which may be represented differently within the computer memory. The data type specifies two things:

1. Permissible range of values that it can store.
2. Memory requirement to store a data type.

C Language provides four basic data types viz. int, char, float and double. Using these, we can store data in simple ways as single elements or we can group them together and use different ways (to be discussed later) to store them as per requirement. The four basic data types are described in the following table 3.1:

Table 3.1: Basic Data Types

DATA TYPE	TYPE OF DATA	MEMORY	RANGE
int	Integer	2 Bytes	– 32,768 to 32,767
char	character	1 Byte	– 128 to 128
float	Floating point number	4 bytes	3.4e – 38 to 3.4e +38
double	Floating point number with higher precision	8 bytes	1.7e – 308 to 1.7e + 308

Memory requirements or size of data associated with a data type indicates the range of numbers that can be stored in the data item of that type.

3.5 DATA TYPE QUALIFIERS

Short, long, signed, unsigned are called the data type qualifiers and can be used with any data type. A ***short int*** requires less space than ***int*** and ***long int*** may require more space than ***int***. If ***int*** and ***short int*** takes 2 bytes, then ***long int*** takes 4 bytes.

Unsigned bits use all bits for magnitude; therefore, this type of number can be larger. For example ***signed int*** ranges from –32768 to +32767 and ***unsigned int*** ranges from 0 to 65,535. Similarly, ***char*** data type of data is used to store a character. It requires 1 byte. ***Signed char*** values range from –128 to 127 and ***unsigned char*** value range from 0 to 255. These can be summarized as follows:

Data type	**Size (bytes)**	**Range**
Short int or int	2	–32768 to 32,767
Long int	4	–2147483648 to 2147483647

Signed int	2	–32768 to 32767
Unsigned int	2	0 to 65535
Signed char	1	–128 to 127
Unsigned char	1	0 to 255

3.6 VARIABLES

Variable is an identifier whose value changes from time to time during execution. It is a named data storage location in your computer's memory. By using a variable's name in your program, you are, in effect, referring to the data stored there. A variable represents a single data item i.e. a numeric quantity or a character constant or a string constant. Note that a value must be assigned to the variables at some point of time in the program which is termed as assignment statement. The variable can then be accessed later in the program. If the variable is accessed before it is assigned a value, it may give garbage value. The data type of a variable doesn't change whereas the value assigned to can change. All variables have three essential attributes:

- the name
- the value
- the memory, where the value is stored.

For example, in the following C program ***a, b, c, d*** are the variables but variable ***e*** is not declared and is used before declaration. After compiling the source code and look what gives?

```
 main( )
{
    int    a, b, c;
    char  d;
           a = 3;
           b = 5;
           c = a + b;
           d = 'a';
           e=d;
           ……….
           ……….
           }
```

After compiling the code, this will generate the message that variable ***e*** not defined.

3.7 DECLARING VARIABLES

Before any data can be stored in the memory, we must assign a name to these locations of memory. For this we make declarations. Declaration associates a group of identifiers with a specific data type. All of them need to be declared before they appear in program statements, else accessing the variables results in junk values or a diagnostic error. The syntax for declaring variables is as follows:

data- type variable-name(s);

For example,

```
int  a;
short int   a, b;
```

```
int  c, d;
long  c, f;
float  r1, r2;
```

3.8 INITIALISING VARIABLES

When variables are declared initial, values can be assigned to them in two ways:

a) Within a Type declaration

The value is assigned at the declaration time.

For example,

```
int      a = 10;
float   b = 0.4 e –5;
char     c = ‘a’;
```

b) Using Assignment statement

The values are assigned just after the declarations are made.

For example,

```
a = 10;
b = 0.4 e –5;
c = ‘a’;
```

Check Your Progress 1

1) Identify keywords and valid identifiers among the following:

hello	function	day-of-the-week
student_1	max_value	“what”
1_student	int	union

…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………

2) Declare type variables for roll no, total_marks and percentage.

…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………

3) How many bytes are assigned to store for the following?

a) Unsigned character b) Unsigned integer c) Double

…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………

3.9 CONSTANTS

A constant is an identifier whose value can not be changed throughout the execution of a program whereas the variable value keeps on changing. In C there are four basic types of **constants**. They are:

1. Integer constants
2. Floating point constants
3. Character constants
4. String constants

Integer and Floating Point constants are numeric constants and represent numbers.

Rules to form Integer and Floating Point Constants

- No comma or blankspace is allowed in a constant.
- It can be preceded by – (minus) sign if desired.
- The value should lie within a minimum and maximum permissible range decided by the word size of the computer.

3.9.1 Integer Constants

Further, these constant can be classified according to the base of the numbers as:

1. **Decimal integer constants**

   These consist of digits 0 through 9 and first *digit should not be 0.*

   For example,

   1 443 32767
   are valid decimal integer constants.

2. **Invalid Decimal integer Constants**

12 ,45	, not allowed
36.0	Illegal char.
1 010	Blankspace not allowed
10 – 10	Illegal char –
0900	The first digit should not be a zero

3. **Octal integer constants**

   These consist of digits 0 through 7. The first digit must be zero in order to identify the constant as an octal number.

   Valid Octal INTEGER constants are:

   0 01 0743 0777

   Invalid Octal integer constants are:

743	does not begin with 0
0438	illegal character 8
0777.77	illegal char .

4. **Hexadecimal integer constants**

These constants begin *with 0x or OX* and are followed by combination of digits taken from hexadecimal digits 0 to 9, a to f or A to F.

Valid Hexadecimal integer constants are:

OX0 OX1 OXF77 Oxabcd.

Invalid Hexadecimal integer constants are:

OBEF	x is not included
Ox.4bff	illegal char (.)
OXGBC	illegal char G

Maximum values these constants can have are as follows:

Integer constants	**Maximum value**
Decimal integer	32767
Octal integer	77777
Hexadecimal integer	7FFF

Unsigned interger constants: Exceed the ordinary integer by magnitude of 2, they are not negative. A character U or u is prefixed to number to make it unsigned.

Long Integer constants: These are used to exceed the magnitude of ordinary integers and are appended by L.

For example,

50000U	decimal unsigned.
1234567889L	decimal long.
0123456L	otal long.
0777777U	otal unsigned.

3.9.2 Floating Point Constants

What is a base 10 number containing decimal point or an exponent.

Examples of valid floating point numbers are:

0.	1.
000.2	5.61123456
50000.1	0.000741
1.6667E+3	0.006e-3

Examples of Invalid Floating Point numbers are:

1	decimal or exponent required.
1,00.0	comma not allowed.
2E+10.2	exponent is written after integer quantity.
3E 10	no blank space.

A Floating Point number taking the value of 5×10^4 can be represented as:

5000.	5e4
5e+4	5E4
5.0e+4	.5e5

The magnitude of floating point numbers range from 3.4E –38 to a maximum of 3.4E+38, through 0.0. They are taken as double precision numbers. Floating Point constants occupy 2 words = 8 bytes.

3.9.3 Character Constants

This constant is a single character enclosed in apostrophes ‘ ’ .

For example, some of the character constants are shown below:

‘A’, ‘x’, ‘3’, ‘$’

‘\0’ is a null character having value zero.

Character constants have integer values associated depending on the character set adopted for the computer. ASCII character set is in use which uses 7-bit code with 2^7 = 128 different characters. The digits 0-9 are having ASCII value of 48-56 and ‘A’ have ASCII value from 65 and ‘a’ having value 97 are sequentially ordered. For example,

‘A’ has 65, blank has 32

ESCAPE SEQUENCE

There are some non-printable characters that can be printed by preceding them with ‘\’ backslash character. Within character constants and string literals, you can write a variety of **escape sequences**. Each escape sequence determines the code value for a single character. You can use escape sequences to represent character codes:

- you cannot otherwise write (such as \n)
- that can be difficult to read properly (such as \t)
- that might change value in different target character sets (such as \a)
- that must not change in value among different target environments (such as \0)

The following is the list of the escape sequences:

Character	**Escape Sequence**
"	\"
'	\'
?	\?
\	\\
BEL	\a
BS	\b
FF	\f
NL	\n
CR	\r
HT	\t
VT	\v

3.9.4 String Constants

It consists of sequence of characters enclosed within double quotes. For example,

“ red ” “ Blue Sea ” “ 41213*(I+3) ”.

3.10 SYMBOLIC CONSTANTS

Symbolic Constant is a name that substitutes for a sequence of characters or a numeric constant, a character constant or a string constant. When program is compiled each occurrence of a symbolic constant is replaced by its corresponding character sequence. The syntax is as follows:

#define name text

where ***name*** implies symbolic name in caps.
text implies value or the text.

For example,

```
#define   printf   print
#define   MAX   100
#define   TRUE   1
#define   FALSE  0
#define    SIZE    10
```

The # character is used for preprocessor commands. A ***preprocessor*** is a system program, which comes into action prior to Compiler, and it replaces the replacement text by the actual text. This will allow correct use of the statement printf.

Advantages of using Symbolic Constants are:

- They can be used to assign names to values
- Replacement of value has to be done at one place and wherever the name appears in the text it gets the value by execution of the preprocessor. This saves time. if the Symbolic Constant appears 20 times in the program; it needs to be changed at one place only.

Check Your Progress 2

1) Write a preprocessor directive statement to define a constant PI having the value 3.14.
……………………………………………………………………………………
……………………………………………………………………………………

2) Classify the examples into Interger, Character and String constants.

‘A’	0147	0xEFH
077.7	“A”	26.4
“EFH”	‘\r’	abc

……………………………………………………………………………………
……………………………………………………………………………………

3) Name different categories of Constants.
……………………………………………………………………………………
……………………………………………………………………………………

3.11 SUMMARY

To summarize we have learnt certain basics, which are required to learn a computer language and form a basis for all languages. Character set includes alphabets, numeric characters, special characters and some graphical characters. These are used to form words in C language or names or identifiers. Variable are the identifiers, which change their values during execution of the program. Keywords are names with specific meaning and cannot be used otherwise.

We had discussed four basic data types - int, char, float and double. Some qualifiers are used as prefixes to data types like signed, unsigned, short, and long.

The constants are the fixed values and may be either Integer or Floating point or Character or String type. Symbolic Constants are used to define names used for constant values. They help in using the name rather bothering with remembering and writing the values.

3.12 SOLUTIONS / ANSWERS

Check Your Progress 1

1. **Keywords:** int, union
   Valid Identifiers: hello, student_1, max_value

2. int rollno;
   float total_marks, percentage;

3. a) 1 byte b) 2 bytes c) 8 bytes

Check Your Progress 2

1. # define PI 3.14

2. **Integer constant**: 0147
   Character constants: ’A’, ‘\r’
   String constants: ”A”, “EFH”

3.13 FURTHER READINGS

1. The C Programming Language, *Kernighan & Ritchie*, PHI Publication.
2. Computer Science A structured programming approach using C*, Behrouza A. Forouzan, Richard F. Gilberg*, Second Edition, Brooks/Cole, Thomson Learning, 2001.
3. Programming with C*, Gottfried*, Second Edition, Schaum Outlines, Tata Mc Graw Hill, 2003.

UNIT 4 EXPRESSIONS AND OPERATORS

Structure

4.0 INTRODUCTION

In the previous unit we have learnt variables, constants, datatypes and how to declare them in C programming. The next step is to use those variables in expressions. For writing an expression we need operators along with variables. An *expression* is a sequence of operators and operands that does one or a combination of the following:

- specifies the computation of a value
- designates an object or function
- generates side effects.

An *operator* performs an operation (evaluation) on one or more operands. An *operand* is a subexpression on which an operator acts.

This unit focuses on different types of operators available in C including the syntax and use of each operator and how they are used in C.

A computer is different from calculator in a sense that it can solve logical expressions also. Therefore, apart from arithmetic operators, C also contains logical operators. Hence, logical expressions are also discussed in this unit.

4.1 OBJECTIVES

After going through this unit you will be able to:

- write and evaluate arithmetic expressions;
- express and evaluate relational expressions;
- write and evaluate logical expressions;
- write and solve compute complex expressions (containing arithmetic, relational and logical operators), and
- check simple conditions using conditional operators.

4.2 ASSIGNMENT STATEMENT

In the previous unit, we have seen that variables are basically memory locations and they can hold certain values. But, how to assign values to the variables? C provides an assignment operator for this purpose. The function of this operator is to assign the values or values in variables on right hand side of an expression to variables on the left hand side.

The syntax of the assignment expression is as follows:

Variable = constant / variable/ expression;

The data type of the variable on left hand side should match the data type of constant/variable/expression on right hand side with a few exceptions where automatic type conversions are possible. Some examples of assignment statements are as follows:

```
b  = a ;       /* b is assigned the value of a */
b = 5 ;       /* b is assigned the value 5*/
b = a+5;    /* b is assigned the value of expr  a+5 */
```

The expression on the right hand side of the assignment statement can be:

- an arithmetic expression;
- a relational expression;
- a logical expression;
- a mixed expression.

The above mentioned expressions are different in terms of the type of operators connecting the variables and constants on the right hand side of the variable. Arithmetic operators, relational operators and logical operators are discussed in the following sections.

For example,

```
int a;
float b,c ,avg, t;
avg = (b+c) / 2;                /*arithmetic expression */
a = b && c;                      /*logical expression*/
a = (b+c) && (b<c);          /* mixed  expression*/
```

4.3 ARITHMETIC OPERATORS

The basic arithmetic operators in C are the same as in most other computer languages, and correspond to our usual mathematical/algebraic symbolism. The following arithmetic operators are present in C:

Operator	**Meaning**
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modular Division

Some of the examples of algebraic expressions and their C notation are given below:

Expression	**C notation**
$\frac{b * g}{d}$	(b *g) / d
a^3+cd	(a*a*a) + (c*d)

The arithmetic operators are all binary operators i.e. all the operators have two operands. The integer division yields the integer result. For example, the expression 10/3 evaluates to 3 and the expression 15/4 evaluates to 3. C provides the modulus operator, %, which yields the reminder after integer division. The modulus operator is an integer operator that can be used only with integer operands. The expression x%y yields the reminder after x is divided by y. Therefore, 10%3 yields 1 and 15%4 yields 3. An attempt to divide by zero is undefined on computer system and generally results in a run- time error. Normally, Arithmetic expressions in C are written in straight-line form. Thus 'a divided by b' is written as a/b.

The operands in arithmetic expressions can be of integer, float, double type. In order to effectively develop C programs, it will be necessary for you to understand the rules that are used for implicit conversation of floating point and integer values in C.

They are mentioned below:

- An arithmetic operator between an integer and integer always yields an integer result.
- Operator between float and float yields a float result.
- Operator between integer and float yields a float result.

If the data type is double instead of float, then we get a result of double data type.

For example,

Operation	**Result**
5/3	1
5.0/3	1.3
5/3.0	1.3
5.0/3.0	1.3

Parentheses can be used in C expression in the same manner as algebraic expression For example,

```
a * (b + c).
```

It may so happen that the type of the expression and the type of the variable on the left hand side of the assignment operator may not be same. In such a case the value for the expression is promoted or demoted depending on the type of the variable on left hand side of = (assignment operator). For example, consider the following assignment statements:

```
int   i;
float b;
i = 4.6;
b = 20;
```

In the first assignment statement, float (4.6) is demoted to int. Hence *i* gets the value 4. In the second statement int (20) is promoted to float, *b* gets 20.0. If we have a complex expression like:

```
float   a, b, c;
int   s;
s = a * b / 5.0 * c;
```

Where some operands are integers and some are float, then int will be promoted or demoted depending on left hand side operator. In this case, demotion will take place since s is an integer.

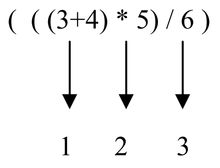
The rules of arithmetic precedence are as follows:

1. Parentheses are at the “highest level of precedence”. In case of nested parenthesis, the innermost parentheses are evaluated first.

For example,

(((3+4)*5)/6)

The order of evaluation is given below.

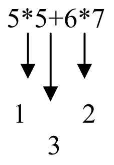


2. Multiplication, Division and Modulus operators are evaluated next. If an expression contains several multiplication, division and modulus operators, evaluation proceeds from left to right. These three are at the same level of precedence.

For example,

5*5+6*7

The order of evaluation is given below.

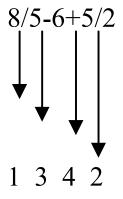


3. Addition, subtraction are evaluated last. If an expression contains several addition and subtraction operators, evaluation proceeds from left to right. Or the associativity is from left to right.

For example,

8/5-6+5/2

The order of evaluation is given below.



Apart from these binary arithmetic operators, C also contains two unary operators referred to as increment (++) and decrement (--) operators, which we are going to be discussed below:
The two-unary arithmetic operators provided by C are:

- ***Increment operator* (++)**
- ***Decrement operator* (- -)**

The increment operator increments the variable by one and decrement operator decrements the variable by one. These operators can be written in two forms i.e. before a variable or after a variable. If an ***increment / decrement*** operator is written before a variable, it is referred to as ***preincrement / predecrement*** operators and if it is written after a variable, it is referred to as ***post increment / postdecrement*** operator.

For example,

a++ or ++a is equivalent to a = a+1 and
a-- or - -a is equivalent to a = a -1

The importance of ***pre*** and ***post*** operator occurs while they are used in the expressions. ***Preincrementing (Predecrementing)*** a variable causes the variable to be incremented (decremented) by 1, then the new value of the variable is used in the expression in which it appears***.*** ***Postincrementing (postdecrementing)*** the variable causes the current value of the variable is used in the expression in which it appears, then the variable value is incremented (decrement) by 1.

The explanation is given in the table below:

Expression	**Explanation**
++a	Increment a by 1, then use the new value of a
a++	Use value of a, then increment a by 1
--b	Decrement b by 1, then use the new value of b
b--	Use the current value of b, then decrement by 1

The precedence of these operators is right to left. Let us consider the following examples:

```
int a = 2, b=3;
int c;
c = ++a – b- -;
printf (“a=%d, b=%d,c=%d\n”,a,b,c);
```

OUTPUT

```
a = 3, b = 2, c = 0.
```

Since the precedence of the operators is right to left, first b is evaluated, since it is a post decrement operator, current value of b will be used in the expression i.e. 3 and then b will be decremented by 1.Then, a preincrement operator is used with a, so first a is incremented to 3. Therefore, the value of the expression is evaluated to 0.

Let us take another example,

```
int  a = 1, b = 2, c = 3;
int k;
```

```
k = (a++)*(++b) + ++a - --c;
printf("a=%d,b=%d, c=%d, k=%d",a,b,c,k);
```

OUTPUT

```
a = 3, b = 3, c = 2, k = 6
```

The evaluation is explained below:

```
k  =  (a++) * (++b)+ ++a  - --c
   =  (a++) * (3) + 2 - 2     step1
   =  (2) * (3) + 2 - 2       step2
   =  6                       final result
```

Check Your Progress 1

1. Give the C expressions for the following algebraic expressions:

   i) $\frac{a*4c^2 - d}{m+n}$

   ii) $ab - (e+f)\frac{4}{c}$

   ………………………………………………………………………………………

   ………………………………………………………………………………………

2. Give the output of the following C code:

```
main()
{
   int a=2,b=3,c=4;
   k = ++b +  --a*c + a;
   printf("a= %d b=%d c=%d k=%d\n",a,b,c,k);
}
```

   ………………………………………………………………………………………

   ………………………………………………………………………………………

3. Point out the error:

```
Exp = a**b;
```

   ………………………………………………………………………………………

   ………………………………………………………………………………………

4.4 RELATIONAL OPERATORS

Executable C statements either perform actions (such as calculations or input or output of data) or make decision. Using relational operators we can compare two variables in the program. The C relational operators are summarized below, with their meanings. Pay particular attention to the equality operator; it consists of two equal signs, not just one. This section introduces a simple version of C's **if** control structure that allows a program to make a decision based on the result of some condition. If the condition is true then the statement in the body of if statement is executed else if the condition is false, the statement is not executed. Whether the body statement is executed or not, after the if structure completes, execution proceeds with the next statement after the if structure. Conditions in the **if** structure are formed with the relational operators which are summarized in the Table 4.1.

Table 1: Relational Operators in C

Relational Operator	Condition	Meaning
==	x==y	x is equal to y
!=	x!=y	x is not equal to y
<	x<y	x is less than y
<=	x<=y	x is less than or equal to y
>	x>y	x is greater than y
>=	x>=y	x is greater or equal to y

Relational operators usually appear in statements which are inquiring about the truth of some particular relationship between variables. Normally, the relational operators in C are the operators in the expressions that appear between the parentheses.
For example,

(i) if (thisNum < minimumSoFar) minimumSoFar = thisNum

(ii) if (job == Teacher) salary == minimumWage

(iii) if (numberOfLegs != 8) thisBug = insect

(iv) if (degreeOfPolynomial < 2) polynomial = linear

Let us see a simple C program containing the If statement (will be introduced in detail in the next unit). It displays the relationship between two numbers read from the keyboard.

Example: 4.1

```
/*Program to find relationship between two numbers*/

#include <stdio.h>
main ( )
{

int a, b;
printf ( “Please enter two integers: ”);
scanf (“%d%d”, &a, &b);
if (a <= b)
printf (“ %d <= %d\n”,a,b);
else
printf (“%d > %d\n”,a,b);
}
```

OUTPUT

```
Please enter two integers: 12 17
12 <= 17
```

We can change the values assigned to a and b and check the result.

4.5 LOGICAL OPERATORS

Logical operators in C, as with other computer languages, are used to evaluate expressions which may be true or false. Expressions which involve logical operations are evaluated and found to be one of two values: **true or false**. So far we have studied simple conditions. If we want to test multiple conditions in the process of making a decision, we have to perform simple tests in separate IF statements(will be introduced in detail in the next unit). C provides logical operators that may be used to form more complex conditions by combining simple conditions.

The logical operators are listed below:

Operator	**Meaning**
&&	Logical AND
||	Logical OR
!	Logical NOT

Thus logical operators (AND and OR) combine two conditions and logical NOT is used to negate the condition i.e. if the condition is true, NOT negates it to false and vice versa.Let us consider the following examples:

(i) Suppose the grade of the student is ‘B’ only if his marks lie within the range 65 to 75,if the condition would be:

```
if ((marks >=65) && (marks <= 75))
printf (“Grade is B\n”);
```

(ii) Suppose we want to check that a student is eligible for admission if his PCM is greater than 85% or his aggregate is greater than 90%, then,

```
if ((PCM >=85) ||(aggregate >=90))
printf (“Eligible for admission\n”);
```

Logical negation (!) enables the programmer to reverse the meaning of the condition. Unlike the && and || operators, which combines two conditions (and are therefore Binary operators), the logical negation operator is a unary operator and has one single condition as an operand. Let us consider an example:

```
if !(grade==’A’)
printf (“the next grade is %c\n”, grade);
```

The parentheses around the condition grade==A are needed because the logical operator has higher precedence than equality operator. In a condition if all the operators are present then the order of evaluation and associativity is provided in the table. The truth table of the logical AND (&&), OR (||) and NOT (!) are given below.

These table show the possible combinations of zero (false) and nonzero (true) values of x (expression1) and y (expression2) and only one expression in case of NOT operator. The following table 4.2 is the truth table for && operator.

Table 4. 2: Truth table for && operator

x	**y**	**x&&y**
zero	zero	0
Non zero	zero	0
zero	Non zero	0
Non zero	Non zero	1

The following table 4.3 is the truth table for || operator.

Table 4.3: Truth table for || operator

x	y	x || y
zero	zero	0
Non zero	zero	1
zero	Non zero	1
Non zero	Non zero	1

The following table 4.4 is the truth table for ! operator.

Table 4.4: Truth table for ! operator

x	! x
zero	1
Non zero	0

The following table 4.5 shows the operator precedence and associativity

Table 4.5: (Logical operators precedence and associativity)

Operator	Associativity
!	Right to left
&&	Left to right
||	Left to right

4.6 COMMA AND CONDITIONAL OPERATORS

Conditional Operator

C provides an called as the conditional operator (**?:**) which is closely related to the **if/else** structure. The conditional operator is C's only ternary operator - it takes three operands. The operands together with the conditional operator form a conditional expression. The first operand is a condition, the second operand represents the value of the entire conditional expression it is the condition is true and the third operand is the value for the entire conditional expression if the condition is false.

The syntax is as follows:

(condition)? (expression1): (expression2);

If condition is true, expression1 is evaluated else expression2 is evaluated. Expression1/Expression2 can also be further conditional expression i.e. the case of nested if statement (will be discussed in the next unit).

Let us see the following examples:

(i) x= (y<20) ? 9: 10;
This means, if (y<20), then x=9 else x=10;

(ii) printf (“%s\n”, grade>=50? “Passed”: “failed”);
The above statement will print “passed” grade>=50 else it will print “failed”

(iii) (a>b) ? printf (“a is greater than b \n”): printf (“b is greater than a \n”);

If a is greater than b, then first printf statement is executed else second printf statement is executed.

Comma Operator

A comma operator is used to separate a pair of expressions. A pair of expressions separated by a comma is evaluated left to right, and the type and value of the result are the value of the type and value of the right operand. All side effects from the evaluation of the left operand are completed before beginning evaluation of the right operand. The left side of comma operator is always evaluated to void. This means that the expression on the right hand side becomes the value of the total comma-separated expression. For example,

```
x = (y=2, y - 1);
```

first assigns y the value 2 and then x the value 1. Parenthesis is necessary since comma operator has lower precedence than assignment operator.

Generally, comma operator (,) is used in the for loop (will be introduced in the next unit)

For example,

```
for (i = 0,j = n;i<j; i++,j--)
{
  printf (“A”);
}
```

In this example **for** is the looping construct (discussed in the next unit). In this loop, i = 0 and j = n are separated by comma (,) and i++ and j—are separated by comma (,). The example will be clear to you once you have learnt for loop (will be introduced in the next unit).

Essentially, the comma causes a sequence of operations to be performed. When it is used on the right hand side of the assignment statement, the value assigned is the value of the last expression in the comma-separated list.

Check Your Progress 2

1. Given a=3, b=4, c=2, what is the result of following logical expressions:
   (a < --b) && (a==c)

   …………………………………………………………………………………………

   …………………………………………………………………………………………

2. Give the output of the following code:

```
main()
{
  int a=10, b=15,x;
```

```
    x = (a<b)?++a:++b;
    printf("x=%d a=%d b=%d\n",x,a,b);
  }
```

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

3. What is the use of comma operator?

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

4.7 TYPE CAST OPERATOR

We have seen in the previous sections and last unit that when constants and variables of different types are mixed in an expression, they are converted to the same type. That is automatic type conversion takes place. The following type conversion rules are followed:

1. All chars and **short ints** are converted to **ints**. All floats are converted to doubles.

2. In case of binary operators, if one of the two operands is a **long double**, the other operand is converted to **long double**,

    else if one operand is **double**, the other is converted to **double**,
    else if one operand is **long**, the other is converted to **long**,
    else if one operand is **unsigned**, the other is converted to **unsigned**,

C converts all operands "up" to the type of largest operand (largest in terms of memory requirement for e.g. **float** requires 4 bytes of storage and **int** requires 2 bytes of storage so if one operand is **int** and the other is **float**, **int** is converted to **float**).

All the above mentioned conversions are automatic conversions, but what if **int** is to be converted to **float.** It is possible to force an expression to be of specific type by using operator called a ***cast***. The syntax is as follows:

(type) expression

where *type* is the standard C data type. For example, if you want to make sure that the expression a/5 would evaluate to type **float** you would write it as

```
( float ) a/5
```

cast is an unary operator and has the same precedence as any other unary operator. The use of ***cast*** operator is explained in the following example:

```
main()
{
   int   num;
   printf("%f %f %f\n", (float)num/2, (float)num/3, float)num/3);
 }
```

Tha ***cast*** operator in this example will ensure that fractional part is also displayed on the screen.

4.8 SIZE OF OPERATOR

C provides a compile-time unary operator called ***sizeof*** that can be used to compute the size of any object. The expressions such as:

sizeof object and ***sizeof(type name)***

result in an unsigned integer value equal to the size of the specified object or type in bytes. Actually the resultant integer is the number of bytes required to store an object of the type of its operand. An object can be a variable or array or structure. An array and structure are data structures provided in C, introduced in latter units. A type name can be the name of any basic type like **int** or **double** or a derived type like a structure or a pointer.

For example,

```
sizeof(char) = 1bytes
sizeof(int) = 2 bytes
```

4.9 C SHORTHAND

C has a special shorthand that simplifies coding of certain type of assignment statements. For example:

```
a = a+2;
```

can be written as:

```
a += 2;
```

The operator +=tells the compiler that a is assigned the value of a + 2;
This shorthand works for all binary operators in C. The general form is:

variable operator = variable / constant / expression

These operators are listed below:

Operators	**Examples**	**Meaning**
+=	a+=2	a=a+2
-=	a-=2	a=a-2
=	**a***=2	a = a*2
/=	a/=2	a=a/2
%=	a%=2	a=a%2

Operators	**Examples**	**Meaning**
&&=	a&&=c	a=a&&c
||=	a||=c	a=a||c

4.10 PRIORITY OF OPERATORS

Since all the operators we have studied in this unit can be used together in an expression, C uses a certain hierarchy to solve such kind of mixed expressions. The hierarchy and associatively of the operators discussed so far is summarized in Table 6. The operators written in the same line have the same priority. The higher precedence operators are written first

Table 4.6: Precedence of the operators

Operators	Associativity
( )	Left to right
! ++ -- (*type*) sizeof	Right to left
/ %	Left to right
+ -	Left to right
< <= > >=	Left to right
== !=	Left to right
&&	Left to right
||	Left to right
?:	Right to left
= += -= *= /= %= &&= ||=	Right to left
,	Left to right

Check Your Progress 3

1. Give the output of the following C code:

```
main( )
{
    int a,b=5;
    float f;

     a=5/2;
     f=(float)b/2.0;
     (a<f)? b=1:b=0;
     printf("b = %d\n",b);
}
```

…………………………………………………………………………………………

…………………………………………………………………………………………

…………………………………………………………………………………………

2. What is the difference between && and &. Explain with an example.

…………………………………………………………………………………………

…………………………………………………………………………………………

…………………………………………………………………………………………

3. Use of Bit Wise operators makes the execution of the program.

…………………………………………………………………………………………

…………………………………………………………………………………………

…………………………………………………………………………………………

4.11 SUMMARY

In this unit, we discussed about the different types of operators, namely arithmetic, relational, logical present in C and their use. In the following units, you will study how these are used in C's other constructs like control statements, arrays etc.

This unit also focused on type conversions. Type conversions are very important to understand because sometimes a programmer gets unexpected results (logical error) which are most often caused by type conversions in case user has used improper types or if he has not type cast to desired type.

This unit also referred to C shorthand. C is referred to as a compact language which is because lengthy expressions can be written in short form. Conditional operator is one of the examples, which is the short form of writing the if/else construct (next unit). Also increment/decrement operators reduce a bit of coding when used in expressions.

Since Logical operators are used further in all types of looping constructs and if/else construct (in the next unit), they should be thoroughly understood.

4.12 SOLUTIONS / ANSWERS

Check Your Progress 1

1. C expression would be

   i) ((a*4*c*c)-d)/(m+n)
   ii) a*b-(e+f)*4/c

2. The output would be:
   a=1 b=4 c=4 k=10

3. There is no such operator as **.

Check Your Progress 2

1. The expression is evaluated as under:

```
(3 < - -4) && (3== 2)
 (3 < 3) &&  (3==2)
 0  && 0
 0
```

   Logical false evaluates to 0 and logical true evaluates to 1.

2. The output would be as follows:

   x=11, a=11, b=16

3. Comma operator causes a sequence of operators to be performed.

Check Your Progress 3

1. Here a will evaluate to 2 and f will evaluate to 2.5 since type cast operator is used in the latter so data type of b changes to float in an expression. Therefore, output would be b=1.

2. && operator is a logical and operator and & is a bit wise and operator. Therefore, && operator always evaluates to true or false i.e 1 or 0 respectively while & operator evaluates bit wise so the result can be any value. For example:

   2 && 5 => 1(true)
   2 & 5 => 0(bit-wise anding)

3. Use of Bit Wise operators makes the execution of the program faster.

4.13 FURTHER READINGS

1. The C Programming Language*, Kernighan & Richie,* PHI Publication**.**
2. Computer Science A structured programming approach using C*, Behrouza A. Forouzan, Richard F. Gilberg, Second Edition, Brooks/Cole*, Thomson Learning, 2001.
3. Programming with C*,* Second Edition, *Byron Gottfried*, Schaum Outline, Tata Mc Graw Hill, 2003.

UNIT 5 DECISION AND LOOP CONTROL STATEMENTS

Structure

5.0 INTRODUCTION

A *program* consists of a number of statements to be executed by the computer. Not many of the programs execute all their statements in sequential order from beginning to end as they appear within the program. A *C program* may require that a logical test be carried out at some particular point within the program. One of the several possible actions will be carried out, depending on the outcome of the *logical test*. This is called ***Branching***. In the ***Selection*** process, a set of statements will be selected for execution, among the several sets available. Suppose, if there is a need of a group of statements to be executed repeatedly until some logical condition is satisfied, then ***looping*** is required in the program. These can be carried out using various control statements.

These ***Control statements*** determine the "*flow of control*" in a program and enable us to specify the order in which the various instructions in a program are to be executed by the computer. Normally, high level procedural programming languages require three basic control statements:

- Sequence instruction
- Selection/decision instruction
- Repetition or Loop instruction

Sequence instruction means executing one instruction after another, in the order in which they occur in the source file. This is usually built into the language as a default action, as it is with C. If an instruction is not a control statement, then the next instruction to be executed will simply be the next one in sequence.

Selection means executing different sections of code depending on a specific condition or the value of a variable. This allows a program to take different courses of action depending on different conditions. C provides three selection structures.

- *if*
- *if...else*
- *switch*

Repetition/Looping means executing the same section of code more than once. A section of code may either be executed a fixed number of times, or while some condition is true. C provides three looping statements:

- *while*
- *do…while*
- *for*

This unit introduces you the decision and loop control statements that are available in C programming language along with some of the example programs.

5.1 OBJECTIVES

After going through this unit you will be able to:

- work with different control statements;
- know the appropriate use of the various control statements in programming;
- transfer the control from within the loops;
- use the ***goto***, ***break*** and ***continue*** statements in the programs; and
- write programs using branching, looping statements.

5.2 DECISION CONTROL STATEMENTS

In a C program, a decision causes a one-time jump to a different part of the program, depending on the value of an expression. Decisions in C can be made in several ways. The most important is with the ***if...else*** statement, which chooses between two alternatives. This statement can be used without the ***else***, as a simple ***if*** statement. Another decision control statement, ***switch***, creates branches for multiple alternative sections of code, depending on the value of a single variable.

5.2.1 The *if* Statement

It is used to execute an *instruction* or sequence/*block of instructions* only if a *condition* is fulfilled. In ***if*** statements, expression is evaluated first and then, depending on whether the value of the expression (relation or condition) is "*true*" or "*false*", it transfers the control to a particular statement or a group of statements.

Different forms of implementation *if*-statement are:

- Simple *if* statement
- *If-else* statement
- *Nested if-else* statement
- *Else if* statement

Simple *if* statement

It is used to execute an instruction or block of instructions only if a condition is fulfilled.

The syntax is as follows:

```
if (condition)
    statement;
```

where c*ondition* is the expression that is to be evaluated. If this *condition* is ***true***, *statement* is executed. If it is ***false***, *statement* is ignored (not executed) and the program continues on the next instruction after the conditional statement.

This is shown in the Figure 5.1 given below:

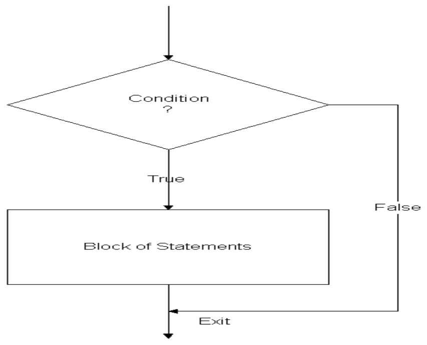


Figure 5.1: Simple *if* statement

If we want more than one statement to be executed, then we can specify a block of statements within the curly bracets { }. The syntax is as follows:

```
if (condition)
   {
     block of statements;
  }
```

Example 5.1

Write a program to calculate the net salary of an employee, if a tax of 15% is levied on his gross-salary if it exceeds Rs. 10,000/- per month.

```
/*Program to calculate the net salary of an employee */

#include <stdio.h>
main( )
{
float gross_salary, net_salary;

printf("Enter gross salary of an employee\n");
scanf("%f ",&gross_salary );

if (gross_salary <10000)
       net_salary= gross_salary;
if (gross_salary >= 10000)
       net_salary = gross_salary- 0.15*gross_salary;

printf("\nNet salary is Rs.%.2f\n", net_salary);
}
```

OUTPUT

```
Enter gross salary of an employee
9000
Net salary is Rs.9000.00

Enter gross salary of any employee
10000
Net salary is Rs. 8500.00
```

If … else statement

If…else statement is used when a different sequence of instructions is to be executed depending on the logical value *(True / False)* of the condition evaluated.

Its form used in conjunction with *if* and the syntax is as follows:

```
if (condition)
        Statement _1;
  else
        Statement_ 2;
statement_3;
```

Or

```
if (condition)
    {
    Statements_1_Block;
    }
else
   {
   Statements_2_Block;
   }
Statements _3_Block;
```

If the *condition* is ***true***, then the sequence of statements (S*tatements_1_Block)* executes; otherwise the *Statements_2_Block* following the *else* part of *if-else* statement will get executed. In both the cases, the control is then transferred to *Statements_3* to follow sequential execution of the program.
This is shown in figure 5.2 given below:

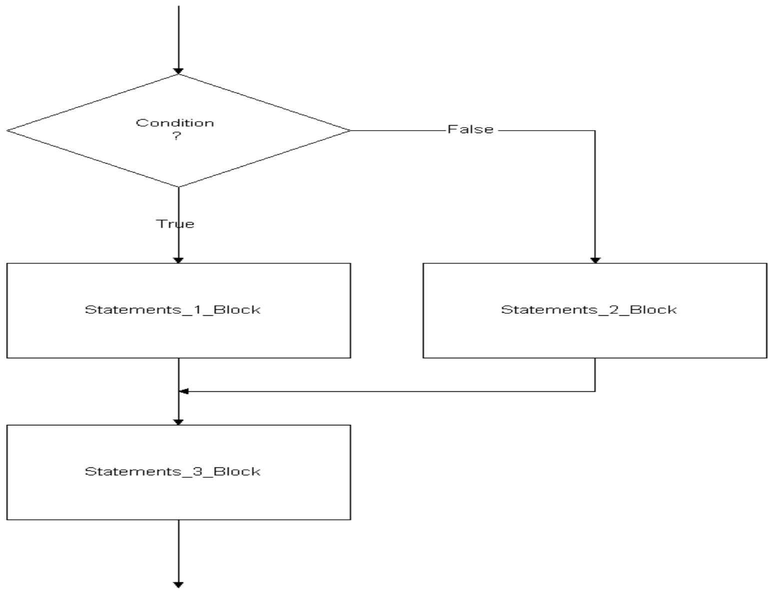


Figure 5.2: *If…else* statement

Let us consider a program to illustrate *if…else* statement,

Example 5.2

Write a program to print whether the given number is even or odd.

```
/* Program to print whether the given number is even or odd*/
#include <stdio.h>
main ( )
{
int x;
printf("Enter a number:\n");
scanf("%d",&x);
if (x % 2 == 0)
        printf("\nGiven number is even\n");
else
        printf("\nGiven number is odd\n");
}
```

OUTPUT

```
Enter a number:
6
Given number is even


Enter a number
7
Given number is odd
```

Nested *if…else* statement

In *nested if… else statement*, an entire *if...else* construct is written within either the body of the ***if*** statement or the body of an *else* statement. The syntax is as follows:

```
if (condition_1)
  {
    if (condition_2)
      {
        Statements_1_Block;
      }

   else
      {
        Statements_2_Block;
      }
   }

else
     {
       Statements_3_Block;
     }
Statement_4_Block;
```

Here, *condition_1* is evaluated. If it is ***false*** then *Statements_3_Block* is executed and is followed by the execution of *Statements_4_Block*, otherwise if *condition_1* is ***true,*** then *condition_2* is evaluated. *Statements_1_Block* is executed when *condition_2* is ***true*** otherwise *Statements_2_Block* is executed and then the control is transferred to *Statements_4_Block*.

This is shown in the figure 5.3 given in the next page:

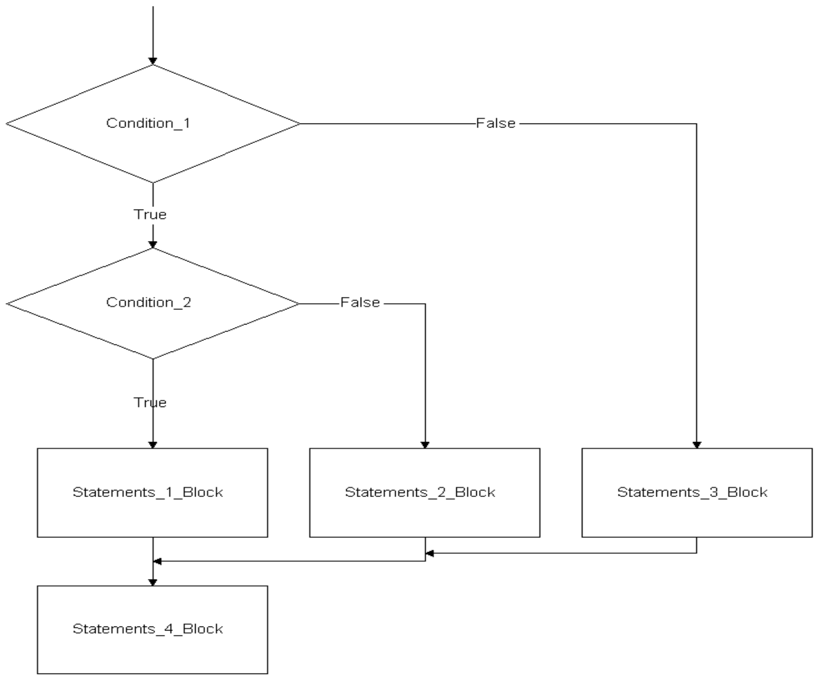


Figure 5.3: Nested *if…else* statement

Let us consider a program to illustrate Nested if…else statement,

Example 5.3

Write a program to calculate an Air ticket fare after discount, given the following conditions:

- If passenger is below 14 years then there is 50% discount on fare
- If passenger is above 50 years then there is 20% discount on fare
- If passenger is above 14 and below 50 then there is 10% discount on fare.

```
/* Program to calculate an Air ticket fare after discount */

#include <stdio.h>
main( )
{
int age;
float fare;
printf("\n Enter the age of passenger:\n");
scanf("%d",&age);
printf("\n Enter the Air ticket fare\n");
scanf("%f",&fare);
if (age < 14)
    fare = fare - 0.5 * fare;
else
    if (age <= 50)
      {
        fare = fare - 0.1 * fare;
      }
        else
      {
        fare = fare - 0.2 * fare;
      }
printf("\n Air ticket fare to be charged after discount is %.2f",fare);
}
```

OUTPUT

```
Enter the age of passenger
12
Enter the Air ticket fare
2000.00
Air ticket fare to be charged after discount is 1000.00
```

Else if statement

To show a multi-way decision based on several conditions, we use the ***else if*** statement. This works by cascading of several comparisons. As soon as one of the conditions is true, the statement or block of statements following them is executed and no further comparisons are performed. The syntax is as follows:

```
if (condition_1)
    {
    Statements_1_Block;
    }
    else if (condition_2)
        {
            Statements_2_Block;
        }
              ------------
                      else if (condition_n)
                         {
                             Statements_n_Block;
                         }
else
    Statements_x;
```

Here, the *conditions* are evaluated in order from top to bottom. As soon as any condition evaluates to *true*, then the statement associated with the given condition is executed and control is transferred to *Statements_x* skipping the rest of the conditions following it. But if all conditions evaluate *false*, then the statement following final ***else*** is executed followed by the execution of *Statements_x*. This is shown in the figure 5.4 given below:

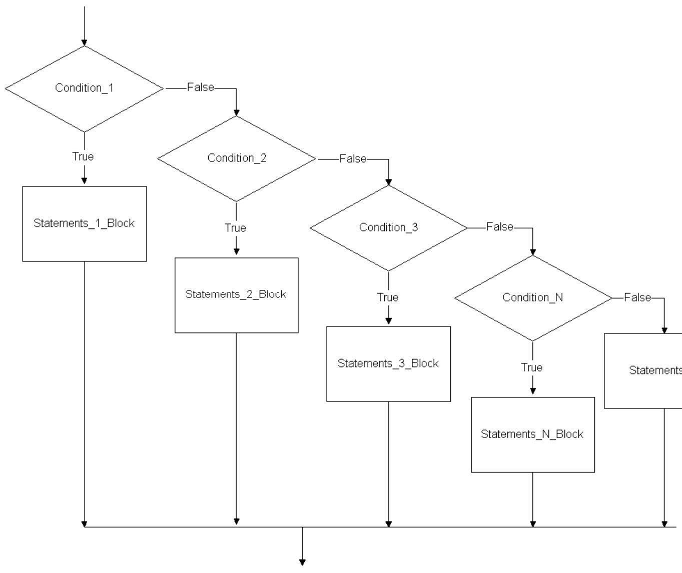


Figure 5.4: ***Else if*** **statement**

Let us consider a program to illustrate *Else if* statement,

Example 5.4

Write a program to award grades to students depending upon the criteria mentioned below:

- Marks less than or equal to 50 are given "D" grade
- Marks above 50 but below 60 are given "C" grade
- Marks between 60 to 75 are given "B" grade
- Marks greater than 75 are given "A" grade.

```
/* Program to award grades */
#include <stdio.h>
main()
{
int result;
printf("Enter the total marks of a student:\n");
scanf("%d",&result);
if (result <= 50)
            printf("Grade D\n");
                 else if (result <= 60)
                        printf("Grade C\n");
                     else if (result <= 75)
                            printf("Grade B\n");
                               else
                                    printf("Grade A\n");
}
```

OUTPUT

```
Enter the total marks of a student:
80
Grade A
```

Check Your Progress 1

1. Find the output for the following program:

```
#include <stdio.h>
main()
{
            int a=1, b=1;
            if(a==0)
                 if(b==0)
                            printf("HI");
                      else
            printf("Bye");
}
```

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

2. Find the output for the following program:

```
#include <stdio.h>
main()
{
```

```
                int a,b=0;
                if (a=b=1)
                        printf("hello");
                else
                        printf("world");
        }
```

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

5.2.2 The *Switch* Statement

Its objective is to check several possible constant values for an expression, something similar to what we had studied in the earlier sections, with the linking of several ***if*** and ***else if*** statements. When the actions to be taken depending on the value of control variable, are large in number, then the use of control structure ***Nested if…else*** makes the program complex. There ***switch*** statement can be used. Its form is the following:

```
switch (expression){
        case expression 1:
                        block of instructions 1
                        break;
        case expression 2:
                        block of instructions 2
                        break;
        .
        .
        default:
              default block of instructions
        }
```

It works in the following way: **switch** evaluates expression and checks if it is equivalent to *expression1*. If it is, it executes *block of instructions 1* until it finds the **break** keyword, moment at finds the control will go to the end of the *switch*. If *expression* was not equal to *expression 1* it will check whether *expression* is equivalent to *expression 2*. If it is, it will execute *block of instructions 2* until it finds the **break** keyword.

Finally, if the value of *expression* has not matched any of the previously specified constants (you may specify as many **case** statements as values you want to check), the program will execute the instructions included in the **default:** section, if it exists, as it is an optional statement.

Let us consider a program to illustrate *Switch* statement,

Example 5.5

Write a program that performs the following, depending upon the choice selected by the user.

i). calculate the square of number if choice is 1
ii). calculate the cube of number if choice is 2 and 4
iii). calculate the cube of the given number if choice is 3
iv). otherwise print the number as it is

```
main()
{
int choice,n;
printf("\n Enter any number:\n ");
scanf("%d",&n);
printf("Choice is as follows:\n\n");
printf("1. To find square of the number\n");
printf("2. To find square-root of the number\n");
printf("3. To find cube of a number\n");
printf("4. To find the square-root of the number\n\n");
printf("Enter your choice:\n");
scanf("%d",&choice);
switch (choice)
{
        case 1 : printf("The square of the number is %d\n",n*n);
                 break;
        case 2 :
        case 4 : printf("The square-root of the given number is %f",sqrt(n));
                 break;
        case 3:  printf(" The cube of the given number is %d",n*n*n);
        default : printf("The number you had given is %d",n);
                  break;
}
}
```

OUTPUT

```
Enter any number:
 4

Choice is as follows:
1. To find square of the number
2. To find square-root of the number\n");
3. To find cube of a number
4. To find the square-root of the number

Enter your choice:
2
The square-root of the given number is 2
```

In this section we had discussed and understood various decision control statements. Next section explains you the various loop control statements in C.

5.3 LOOP CONTROL STATEMENTS

Loop control statements are used when a section of code may either be executed a fixed number of times, or while some condition is true. C gives you a choice of three types of loop statements, *while*, *do- while* and *for*.

- The *while* loop keeps repeating an action until an associated *condition* returns ***false***. This is useful where the programmer does not know in advance how many times the loop will be traversed.
- The *do while* loop is similar, but the c*ondition* is checked after the loop body is executed. This ensures that the loop body is run at least once.
- The *for* loop is frequently used, usually where the loop will be traversed a fixed number of times.

5.3.1 The *While* Loop

When in a program a single statement or a certain group of statements are to be executed repeatedly depending upon certain test condition, then *while statement* is used.

The syntax is as follows:

while (test condition*)*
{
  body_of_the_loop;
}

Here, *test condition* is an expression that controls how long the loop keeps running. Body of the loop is a statement or group of statements enclosed in braces and are repeatedly executed till the value of *test condition* evaluates to *true*. As soon as the *condition* evaluates to ***false***, the control jumps to the first statement following the *while* statement. If condition initially itself is ***false***, the body of the loop will never be executed. *While* loop is sometimes called as *entry-control loop,* as it controls the execution of the body of the loop depending upon the value of the *test condition*. This is shown in the figure 5.5 given below:

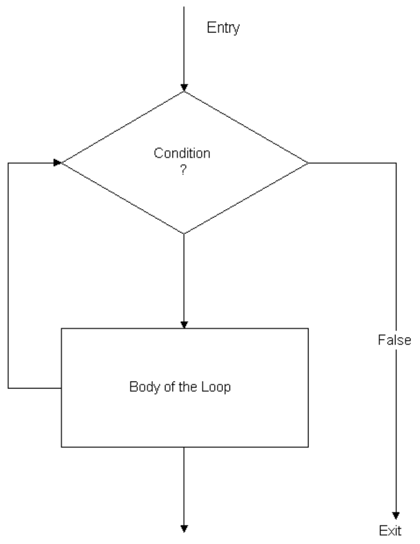


Figure 5.5: The *while* loop statement

Let us consider a program to illustrate *while loop*,

Example 5.6

Write a program to calculate the factorial of a given input natural number.

```
/* Program to calculate factorial of given number */

#include <stdio.h>
#include <math.h>
#include <stdio.h>
main( )
{
int x;
long int fact = 1;
printf(“Enter any number to find factorial:\n”);                    /*read the number*/
scanf(“%d”,&x);
while (x > 0)
     {
          fact = fact * x;           /* factorial calculation*/
          x=x-1;
         }
printf(“Factorial is %ld”,fact);
}
```

OUTPUT

```
Enter any number to find factorial:
4
Factorial is 24
```

Here, *condition* in *while* loop is evaluated and body of loop is repeated until *condition* evaluates to ***false*** i.e., when x becomes zero. Then the control is jumped to first statement following *while* loop and print the value of factorial.

5.3.2 The *do...while* Loop

There is another loop control structure which is very similar to the *while* statement – called as the ***do.. while*** statement. The only difference is that the expression which determines whether to carry on looping is evaluated at the end of each loop. The syntax is as follows:

```
do
{
  statement(s);
} while(test condition);
```

In *do-while* loop, the body of loop is executed at least once before the *condition* is evaluated. Then the loop repeats body as long as *condition* is ***true***. However, in *while* loop, the statement doesn't execute the body of the loop even once, if *condition* is ***false***. That is why *do-while* loop is also called *exit-control loop*. This is shown in the figure 5.6 given below.

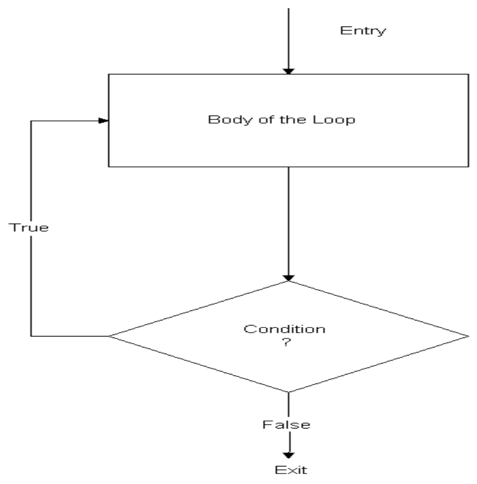


Figure 5.6: The *do…while* statement

Let us consider a program to illustrate do..*while loop*,

Example 5.7

Write a program to print first ten even natural numbers.

```
/* Program to print first ten even natural numbers */
#include <stdio.h>
main()
{
```

```
int i=0;
int j=2;
do   {
          printf("%d",j);
           j =j+2;
           i=i+1;   } while (i<10);      }
```

OUTPUT

```
2 4 6 8 10 12 14 16 18 20
```

5.3.3 The *for* Loop

for statement makes it more convenient to count iterations of a loop and works well where the number of iterations of the loop is known before the loop is entered. The syntax is as follows:

for (*initialization*; *test condition*; *increment or decrement*)
{
    Statement(s);
}

The main purpose is to repeat *statement* while *condition* remains true, like the *while* loop. But in addition, ***for*** provides places to specify an *initialization* instruction and an *increment or decrement of the control variable* instruction. So this loop is specially designed to perform a repetitive action with a counter.

The *for* loop as shown in figure 5.7, works in the following manner:

1. *initialization* is executed. Generally it is an initial value setting for a counter variable. This is executed only once.
2. *condition* is checked, if it is *true* the loop continues, otherwise the loop finishes and *statement* is skipped.
3. S*tatement(s)* is/are executed. As usual, it can be either a single instruction or a block of instructions enclosed within curly brackets { }.
4. Finally, whatever is specified in the *increment or decrement of the control variable* field is executed and the loop gets back to step 2.

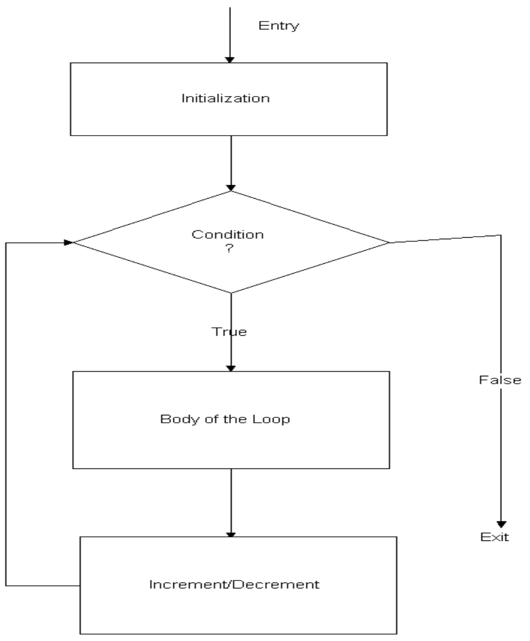


Figure 5.7: The *for* statement

Let us consider a program to illustrate *for loop*,

Example 5.8

Write a program to print first n natural numbers.

```
/* Program to print first n natural numbers */

#include <stdio.h>
main( )
{
int i,n;
printf("Enter  value of n \n");
scanf("%d",&n);
printf("\nThe first %d natural numbers are :\n", n);
for (i=1;i<=n;++i)
   {
     printf("%d",i);
    }
}
```

OUTPUT

```
Enter value of n
6
The first 6 natural numbers are:
1 2 3 4 5 6
```

The three statements inside the braces of a *for* loop usually meant for one activity each, however any of them can be left blank also. More than one control variables can be initialized but should be separated by comma.

Various forms of loop statements can be:

(a) *for(;condition;increment/decrement)*
body;
A blank first statement will mean no initialization.

(b) *for (initialization;condition;)*
body;
A blank last statement will mean no running increment/decrement.

(c) *for (initialization;;increment/decrement)*
body;

A blank second conditional statement means no test condition to control the exit from the loop. So, in the absence of second statement, it is required to test the condition inside the loop otherwise it results in an infinite loop where the control never exits from the loop.

(d) *for (;;increment/decrement)*
body;
Initialization is required to be done before the loop and test condition is checked inside the loop.

(e) *for (initialization;;)*
body;

Test condition and *control variable* increment/decrement is to be done inside the body of the loop.

(f) for (;condition;)
body;
Initialization is required to be done before the loop and control variable increment/decrement is to be done inside the body of the loop.

(g) for (;;;)
body;
Initialization is required to be done before the loop, *test condition* and *control variable* increment/decrement is to be done inside the body of the loop.

5.3.4 The Nested Loops

C allows loops to be *nested*, that is, one loop may be inside another. The program given below illustrates the *nesting* of loops.

Let us consider a program to illustrate *nested loops*,

Example 5.9

Write a program to generate the following pattern given below:

```
1
1    2
1    2    3
1    2    3    4
```

```
/* Program to print the pattern */

#include <stdio.h>
main( )
{
int i,j;
for (i=1;i<=4;++i)
  {
    printf("%d\n",i);
        for(j=1;j<=i;++j)
                printf("%d\t",j);
        }
}
```
Here, an *inner for loop* is written inside the *outer for loop*. For every value of ***i, j*** takes the value from 1 to ***i*** and then value of ***i*** is incremented and next iteration of outer loop starts ranging ***j*** value from 1 to ***i***.

Check Your Progress 2

1. Predict the output :

```
  #include <stdio.h>
  main()
  {
    int i;
        for (i=0;i<=10;i++,printf("%d ",i));
  }
```
………………………………………………………………………………………

………………………………………………………………………………………

2. What is the output?

```
#include <stdio.h>
main( )
{
        int i;
        for(i=0;i<3;i++)
         printf("%d ",i);
}
```

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

3. What is the output for the following program?

```
#include <stdio.h>
main( )
{
  int i=1;
  do
  {
     printf("%d",i);
  }while(i=i-1);
}
```

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

4. Give the output of the following:

```
#include <stdio.h>
main( )
{
   int i=3;
   while(i)
   {
       int x=100;
       printf("\n%d..%d",i,x);
       x=x+1;
       i=i+1;
   }
}
```

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

5.4 THE *goto* STATEMENT

The ***goto*** statement is used to alter the normal sequence of program instructions by transferring the control to some other portion of the program. The syntax is as follows:

goto label;

Here, ***label*** is an identifier that is used to label the statement to which control will be transferred. The targeted statement must be preceded by the unique label followed by colon.

label : statement;

Although *goto* statement is used to alter the normal sequence of program execution but its usage in the program should be avoided. The most common applications are:

i). To branch around statements under certain conditions in place of use of *if-else* statement,
ii). To jump to the end of the loop under certain conditions bypassing the rest of statements inside the loop in place of *continue* statement,
iii). To jump out of the loop avoiding the use of *break* statement.

goto can never be used to jump into the loop from outside and it should be preferably used for forward jump.

Situations may arise, however, in which the ***goto*** statement can be useful. To the possible extent, the use of the ***goto*** statement should generally be avoided.

Let us consider a program to illustrate *goto* and *label* statements.

Example 5.10

Write a program to print first 10 even numbers

```
/* Program to print 10 even numbers */

#include <stdio.h>
main()
{
 int i=2;
 while(1)
       {
         printf("%d ",i);
         i=i+2;
           if (i>=20)
               goto outside;
       }
  outside : printf("over");
  }
```

OUTPUT

```
2 4 6 8 10 12 14 16 18 20 over
```

5.5 THE *break* STATEMENT

Sometimes, it is required to jump out of a loop irrespective of the *conditional test value*. ***Break*** statement is used inside any loop to allow the control jump to the immediate statement following the loop. The syntax is as follows:

break;

When nested loops are used, then ***break*** jumps the control from the loop where it has been used. *Break* statement can be used inside any loop i.e., *while*, *do-while*, *for* and also in *switch* statement.

Let us consider a program to illustrate *break* statement.

Example 5.11

Write a program to calculate the first smallest divisor of a number.

```
/*Program to calculate smallest divisor of a number */

#include <stdio.h>
main( )
{
int div,num,i;
printf("Enter any number:\n");
scanf("%d",&num);
for (i=2;i<=num;++i)
    {
  if ((num % i) == 0)
          {
          printf("Smallest divisor for number %d is %d",num,i);
          break;
          }
    }
}
```

OUTPUT

```
Enter  any  number:
9
Smallest divisor for number 9 is 3
```

In the above program, we divide the input number with the integer starting from 2 onwards, and print the smallest divisor as soon as remainder comes out to be zero. Since we are only interested in first smallest divisor and not all divisors of a given number, so jump out of the *for* loop using *break* statement without further going for the next iteration of *for* loop.

Break is different from *exit*. Former jumps the control out of the loop while exit stops the execution of the entire program.

5.6 THE *continue* STATEMENT

Unlike *break* statement, which is used to jump the control out of the loop, it is sometimes required to skip some part of the loop and to continue the execution with next loop iteration. ***Continue*** statement used inside the loop helps to bypass the section of a loop and passes the control to the beginning of the loop to continue the execution with the next loop iteration. The syntax is as follows:

continue;

Let us see the program given below to know the working of the ***continue*** statement.

Example 5.12

Write a program to print first 20 natural numbers skipping the numbers divisible by 5.

```
/*  Program to print first 20 natural numbers skipping the numbers divisible by 5 */

#include <stdio.h>
main( )
{
          int i;
          for (i=1;i<=20;++i)
          {
```

```
                    if ((i % 5) == 0)
                              continue;
                    printf("%d ",i);
          }
}
```

OUTPUT

```
1 2 3 4 6 7 8 9 11 12 13 14 16 17 18 19
```

Here, the printf statement is bypassed each time when value stored in ***i*** is divisible by 5.

Check Your Progress 3

1. How many times will hello be printed by the following program?

```
    #include <stdio.h>
    main( )
    {
          int i = 5;
          while(i)
            {
          i=i-1;
          if (i==3)
          continue;
          printf("\nhello");
          }
    }
```

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

2. Give the output of the following program segment:

```
     #include <stdio.h>
     main( )
     {
     int num,sum;
     for (num=2,sum=0;;)
        {
       sum = sum + num;
        if (num > 10)
            break;
        num=num+1;
        }
        printf("%d",sum);
     }
```

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

3. What is the output for the following program?

```
     #include <stdio.h>
     main( )
     {
         int i, n = 3;
```

```
            for (i=3;n<=20;++n)
             {
               if (n%i == 0)
               break;
               if (i == n)
                printf("%d\n",i);
             }
      }
```

………………………………………………………………………………………

………………………………………………………………………………………

………………………………………………………………………………………

5.7 SUMMARY

A *program* is usually not limited to a linear sequence of instructions. During its process it may require to repeat execution of a part of code more than once depending upon the requirements or take decisions. For that purpose, C provides *control* and looping statements. In this unit, we had seen the different looping statements provided by C language namely ***while, do…while and for.***

Using ***break*** statement, we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. The ***continue*** statement causes the program to skip the rest of the loop in the present iteration as if the end of the *statement* block would have reached, causing it to jump to the following iteration.

Using the ***goto*** statement, we can make an absolute jump to another point in the program. You should use this feature carefully since its execution ignores any type of nesting limitation. The destination point is identified by a label, which is then used as argument for the *goto* instruction. A *label* is made of a valid identifier followed by a colon (**:**).

5.8 SOLUTIONS / ANSWERS

Check Your Progress 1

1 Nothing

2 hello

Check Your Progress 2

1 1 2 3 4 5 6 7 8 9 10 11

2 0 1 2

3 1 0 2

4 3..100
2..100
1..100
…..
…..
…...
till infinity

Check Your Progress 3

1 4 times

2 65

3 3

5.9 FURTHER READINGS

1. The C programming language, *Brain W. Kernighan, Dennis M. Ritchie*, PHI.
2. Programming with C, Second Edition, *Byron Gottfried*, Tata McGraw Hill, 2003.
3. C,The Complete Reference, Fourth Edition, *Herbert Schildt*, Tata McGraw Hill,
4. 2002.
5. Computer Science: A Structured Programming Approach Using C, Second Edition, *Behrouz A. Forouzan, Richard F. Gilberg,* Brooks/Cole Thomas Learning, 2001.
6. The C Primer, *Leslie Hancock*, *Morris Krieger*, Mc Graw Hill, 1983.

UNIT 6 ARRAYS

Structure

6.0 INTRODUCTION

C language provides four basic data types - *int, char, float and double.* We have learnt about them in Unit 3. These basic data types are very useful; but they can handle only a limited amount of data. As programs become larger and more complicated, it becomes increasingly difficult to manage the data. Variable names typically become longer to ensure their uniqueness. And, the number of variable names makes it difficult for the programmer to concentrate on the more important task of correct coding. Arrays provide a mechanism for declaring and accessing several data items with only one identifier, thereby simplifying the task of data management.

Many programs require the processing of multiple, related data items that have common characteristics like *list* of numbers, marks in a course, or enrolment numbers. This could be done by creating several individual variables. But this is a hard and tedious process. For example, suppose you want to read in five numbers and print them out in reverse order. You could do it the hard way as:

```
main()
{
 int al,a2,a3,a4,a5;
 scanf("%d %d %d %d %d",&a1,&a2,&a3,&a4,&a5);
 printf("%d %d %d %d %d'",a5,a4,a3,a2,a1);
}
```

Does it look good if the problem is to read in 100 or more related data items and print them in reverse order? Of course, the solution is the use of the regular variable names **a1**, **a2** and so on. But to remember each and every variable and perform the operations on the variables is not only tedious a job and disadvantageous too. One common organizing technique is to use arrays in such situations. An array is a collection of similar kind of data elements stored in adjacent memory locations and are referred to by a single array-name. In the case of C, you have to declare and define **array** before it can be used. Declaration and definition tell the compiler the name of the array, the type of each element, and the size or number of elements.To explain it, let us consider to store marks of five students. They can be stored using five variables as follows:

int ar1, ar2, ar3, ar4, ar5;

Now, if we want to do the same thing for 100 students in a class then one will find it difficult to handle 100 variables. This can be obtained by using an array. An array declaration uses its size in [ ] brackets. For above example, we can define an array as:

int ar [100];

where *ar* is defined as an array of size 100 to store marks of integer data-type. Each element of this collection is called an *array-element* and an integer value called the *subscript* is used to denote individual elements of the array. An *ar* array is the collection of 200 consecutive memory locations referred as below:

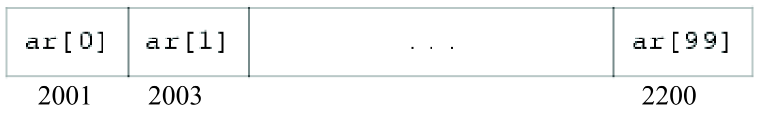


Figure 6.1: Representation of an array

In the above figure, as each integer value occupies 2 bytes, 200 bytes were allocated in the memory.

This unit explains the use of arrays, types of arrays, declaration and initialization with the help of examples.

6.1 OBJECTIVES

After going through this unit you will be able to:

- declare and use arrays of one dimension;
- initialize arrays;
- use subscripts to access individual array elements;
- write programs involving arrays;
- do searching and sorting; and
- handle multi-dimensional arrays.

6.2 ARRAY DECLARATION

Before discussing how to declare an array, first of all let us look at the characteristic features of an array.

- Array is a data structure storing a group of elements, all of which are of the same data type.
- All the elements of an array share the same name, and they are distinguished from one another with the help of an index.
- Random access to every element using a numeric index (subscript).
- A simple data structure, used for decades, which is extremely useful.
- Abstract Data type (ADT) *list* is frequently associated with the array data structure.

The declaration of an array is just like any variable declaration with additional *size* part, indicating the number of elements of the array. Like other variables, arrays must be declared at the beginning of a function.

The declaration specifies the base type of the array, its name, and its size or dimension. In the following section we will see how an array is declared:

6.2.1 Syntax of Array Declaration

Syntax of array declaration is as follows:

data-type array_name [constant-size];

Data-type refers to the type of elements you want to store
Constant-size is the number of elements

The following are some of declarations for arrays:

```
int      char [80];
float    farr [500];
static   int iarr [80];
char     charray [40];
```

There are two restrictions for using arrays in C:

- The amount of storage for a declared array has to be specified at **compile time** before execution. This means that an array has a fixed size.
- The data type of an array applies uniformly to all the elements; for this reason, an array is called a **homogeneous** data structure.

6.2.2 Size Specification

The size of an array should be declared using symbolic constant rather a fixed integer quantity (The subscript used for the individual element is of are integer quantity). The use of a symbolic constant makes it easier to modify a program that uses an array. All reference to maximize the array size can be altered simply by changing the value of the symbolic constant. (Please refer to Unit – 3 for details regarding symbolic constants).

To declare size as 50 use the following symbolic constant, SIZE, defined:

#define SIZE 50

The following example shows how to declare and read values in an array to store marks of the students of a class.

Example 6.1

Write a program to declare and read values in an array and display them.

```
/* Program to read values in an array*/

# include < stdio.h >
# define SIZE 5                          /* SIZE is a symbolic constant */

main ( )
{
int i = 0;                          /* Loop variable */
int stud_marks[SIZE]; /* array declaration */

/* enter the values of the elements */
for( i = 0;i<SIZE;i++)
   {
     printf (“Element no. =%d”,i+1);
    printf(“ Enter the value of the element:”);
```

```
        scanf("%d",&stud_marks[i]);
      }
printf("\nFollowing are the values stored in the corresponding array elements: \n\n");
for( i = 0; i<SIZE;i++)
        {
            printf("Value stored in a[%d] is %d\n"i, stud_marks[i]);
        }
}
```

OUTPUT:

```
Element no.  = 1    Enter the value of  the element = 11
Element no.  = 2    Enter the value of  the element = 12
Element no.  = 3    Enter the value of  the element = 13
Element no.  = 4    Enter the value of  the element = 14
Element no.  = 5    Enter the value of  the element = 15

Following are the values stored in the corresponding array elements:

Value stored in a[0] is 11
Value stored in a[1] is 12
Value stored in a[2] is 13
Value stored in a[3] is 14
Value stored in a[4] is 15
```

6.3 ARRAY INITIALIZATION

Arrays can be initialized at the time of declaration. The initial values must appear in the order in which they will be assigned to the individual array elements, enclosed within the braces and separated by commas. In the following section, we see how this can be done.

6.3.1 Initialization of Array Elements in the Declaration

The values are assigned to individual array elements enclosed within the braces and separated by comma. Syntax of array initialization is as follows:

data type array-name [size] = {val 1, val 2, ...val n};

val 1 is the value for the first array element, *val 2* is the value for the second element, and *val n* is the value for the *n* array element. Note that when you are initializing the values at the time of declaration, then there is no need to specify the size. Let us see some of the examples given below:

```
int digits  [10] = {1,2,3,4,5,6,7,8,9,10};

int digits[ ] = {1,2,3,4,5,6,7,8,9,10};

int vector[5] = {12,-2,33,21,13};

float temperature[10] ={ 31.2, 22.3, 41.4, 33.2, 23.3, 32.3, 41.1, 10.8, 11.3, 42.3};

double width[ ] = { 17.33333456, -1.212121213, 222.191345 };

int height[ 10 ] = { 60, 70, 68, 72, 68 };
```

6.3.2 Character Array Initialisation

The array of characters is implemented as strings in C. Strings are handled differently as far as initialization is concerned. A special character called null character ***' \0 '***, implicitly suffixes every string. When the external or static string character array is assigned a string constant, the size specification is usually omitted and is automatically assigned; it will include the '\0'character, added at end. For example, consider the following two assignment statements:

```
char thing [ 3 ]  =  "TIN";
char thing [  ]   =  "TIN";
```

In the above two statements the assignments are done differently. The first statement is not a string but simply an array storing three characters 'T', 'I' and 'N' and is same as writing:

```
char thing [ 3 ]  =  {'T', 'I', 'N'};
```

whereas, the second one is a four character string TIN\0. The change in the first assignment, as given below, can make it a string.

```
char thing [ 4 ]  =  "TIN";
```

Check Your Progress 1

1. What happens if I use a subscript on an array that is larger than the number of elements in the array?
   ……………………………………………………………………………………
   ……………………………………………………………………………………
2. Give sizes of following arrays.
   a. char carray [ ]= "HELLO";
   b. char carray [5]= "HELLO";
   c. char carray [ ]={ 'H', 'E', 'L', 'L', 'O' };
   ……………………………………………………………………………………
   ……………………………………………………………………………………
3. What happens if an array is used without initializing it?
   ……………………………………………………………………………………
   ……………………………………………………………………………………
4. Is there an easy way to initialize an entire array at once?
   ……………………………………………………………………………………
   ……………………………………………………………………………………
5. Use a *for* loop to total the contents of an integer array called numbers with five elements. Store the result in an integer called TOTAL.
   ……………………………………………………………………………………
   ……………………………………………………………………………………

6.4 SUBSCRIPT

To refer to the individual element in an array, a subscript is used. Refer to the statement we used in the Example 6.1,

```
scanf (" % d", &stud_marks[ i]);
```

Subscript is an integer type constant or variable name whose value ranges from 0 to SIZE - 1 where SIZE is the total number of elements in the array. Let us now see how we can refer to individual elements of an array of size 5:

Consider the following declarations:

```
char country[ ] = “India”;
int stud[ ] = {1, 2, 3, 4, 5};
```

Here both arrays are of size 5. This is because the country is a char array and initialized by a string constant “India” and every string constant is terminated by a null character ‘\0’. And stud is an integer array. country array occupies 5 bytes of memory space whereas stud occupies size of 10 bytes of memory space. The following table: 6.1 shows how individual array elements of *country* and *stud* arrays can be referred:

Table 6.1: Reference of individual elements

Element no.	Subscript	country array		stud array	
		Reference	Value	Reference	Value
1	0	country [0]	‘I’	stud [0]	1
2	1	country [1]	‘n’	stud [1]	2
3	2	country [2]	‘d’	stud [2]	3
4	3	country [3]	‘i’	stud [3]	4
5	4	country [4]	‘a’	stud [4]	5

Example 6.2

Write a program to illustrate how the marks of 10 students are read in an array and then used to find the maximum marks obtained by a student in the class.

```
/* Program to find the maximum marks among the marks of 10 students*/

# include < stdio.h >
# define SIZE 10                    /* SIZE is a symbolic constant */

main ( )
{

int i = 0;
int max = 0;
int stud_marks[SIZE]; /* array declaration */

/* enter the values of the elements */
for( i = 0;i<SIZE;i++)
   {
     printf (“Student no. =%d”,i+1);
     printf(“ Enter the marks out of 50:”);
     scanf(“%d”,&stud_marks[i]);
   }

/* find maximum */
for (i=0;i<SIZE;i ++)
   {
    if (stud_marks[i]>max)
     max = stud_marks[ i ];
    }
printf("\n\nThe maximum of the marks obtained  among all the 10 students is: %d
        ",max);
}
```

OUTPUT

```
Student no. = 1   Enter the marks out of 50: 10
Student no. = 2   Enter the marks out of 50: 17
Student no. = 3   Enter the marks out of 50: 23
Student no. = 4   Enter the marks out of 50: 40
Student no. = 5   Enter the marks out of 50: 49
Student no. = 6   Enter the marks out of 50: 34
Student no. = 7   Enter the marks out of 50: 37
Student no. = 8   Enter the marks out of 50: 16
Student no. = 9   Enter the marks out of 50: 08
Student no. = 10 Enter the marks out of 50: 37

The maximum of the marks obtained among all the 10 students is: 49
```

6.5 PROCESSING THE ARRAYS

For certain applications the assignment of initial values to elements of an array is required. This means that the array be defined globally (extern) or locally as a static array.

Let us now see in the following example how the marks in two subjects, stored in two different arrays, can be added to give another array and display the average marks in the below example.

Example 6.3:

Write a program to display the average marks of each student, given the marks in 2 subjects for 3 students.

```
/* Program to display the average marks of 3 students */

# include < stdio.h >
# define SIZE 3
main()
{
int  i  = 0;
float   stud_marks1[SIZE];         /* subject 1array declaration */
float   stud_marks2[SIZE];         /*subject 2 array declaration */
float   total_marks[SIZE];
float   avg[SIZE];

printf("\n Enter the marks in subject-1 out of 50 marks: \n");
for( i = 0;i<SIZE;i++)
            {
              printf("Student no. =%d",i+1);
              printf(" Enter the marks= ");
              scanf("%f",&stud_marks1[i]);
            }
printf("\n Enter the marks in subject-2 out of 50 marks \n");
    for(i=0;i<SIZE;i++)
            {
                printf("Student no. =%d",i+1);
                printf(" Please enter the marks= ");
               scanf("%f",&stud_marks2[i]);
               }

      for(i=0;i<SIZE;i++)
      {
              total_marks[i]=stud_marks1[i]+ stud_marks2[i];
                     avg[i]=total_marks[i]/2;
                     printf("Student no.=%d, Average= %f\n",i+1, avg[i]);
             }
             }
```

OUTPUT

```
Enter the marks in subject-1out of 50 marks:
Student no. = 1  Enter the marks= 23
Student no. = 2  Enter the marks= 35
Student no. = 3  Enter the marks= 42

Enter the marks in subject-2 out of 50 marks:
Student no. = 1  Enter the marks= 31
Student no. = 2  Enter the marks= 35
Student no. = 3  Enter the marks= 40

Student no. = 1 Average= 27.000000
Student no. = 2 Average= 35.000000
Student no. = 3 Average= 41.000000
```

Let us now write another program to search an element using the linear search.

Example 6.4

Write a program to search an element in a given list of elements using Linear Search.

```
/* Linear Search.*/

# include<stdio.h>
# define SIZE   05
main()
{
int  i  = 0;
int  j;
int   num_list[SIZE];      /* array declaration */

/* enter elements in the following loop */

printf("Enter any 5 numbers: \n");
for(i = 0;i<SIZE;i ++)
      {
           printf("Element no.=%d Value of the element=",i+1);
           scanf("%d",&num_list[i]);
     }
printf ("Enter the element to be searched:");
scanf ("%d",&j);

/* search using linear search */
for(i=0;i<SIZE;i++)
```

```
    {
    if(j == num_list[i])
           {
       printf("The number exists in the list at position: %d\n",i+1);
              break;
           }
    }
}
```

OUTPUT

```
Enter any 5 numbers:
Element no.=1 Value of the element=23
Element no.=2 Value of the element=43
Element no.=3 Value of the element=12
Element no.=4 Value of the element=8
Element no.=5 Value of the element=5
Enter the element to be searched: 8
The number exists in the list at position: 4
```

Example 6.5

Write a program to sort a list of elements using the selection sort method

```
/* Sorting list of numbers using selection sort method*/

#include <stdio.h>
#define SIZE 5

main()
{

int j,min_pos,tmp;
int i;                    /* Loop variable */
int a[SIZE];      /* array declaration */

/* enter the elements  */

for(i=0;i<SIZE;i++)
   {
    printf("Element no.=%d",i+1);
    printf("Value of the element: ");
    scanf("%d",&a[i]);
   }

/* Sorting by descending order*/

for (i=0;i<SIZE;i++)
  {
   min_pos = i;
   for (j=i+1;j<SIZE;j++)
     if (a[j] < a[min_pos])
           min_pos = j;
   tmp = a[i];
   a[i] = a[min_pos];
   a[min_pos] = tmp;
 }
/* print the result */

printf("The array after sorting:\n");
      for(i=0;i<SIZE;i++)
        printf("% d\n",a[i]);
}
```

OUTPUT

```
Element no. = 1 Value of the element: 23
Element no. =2  Value of the element: 11
Element no. = 3 Value of the element: 100
Element no. = 4 Value of the element: 42
Element no. = 5 Value of the element: 50

The array after sorting:
11
23
42
50
100
```

Check Your Progress 2

1. Name the technique used to pass an array to a function.
   …………………………………………………………………………………
   …………………………………………………………………………………
2. Is it possible to pass the whole array to a function?
   …………………………………………………………………………………
   …………………………………………………………………………………
3. List any two applications of arrays.
   …………………………………………………………………………………
   …………………………………………………………………………………

6.6 MULTI-DIMENSIONAL ARRAYS

Suppose that you are writing a chess-playing program. A chessboard is an 8-by-8 grid. What data structure would you use to represent it? You could use an array that has a chessboard-like structure, i.e. a *two-dimensional array*, to store the positions of the chess pieces. Two-dimensional arrays use two indices to pinpoint an individual element of the array. This is very similar to what is called "algebraic notation", commonly used in chess circles to record games and chess problems.

In principle, there is no limit to the number of subscripts (or dimensions) an array can have. Arrays with more than one dimension are called *multi- dimensional arrays*. While humans cannot easily visualize objects with more than three dimensions, representing multi-dimensional arrays presents no problem to computers. In practice, however, the amount of memory in a computer tends to place limits on the size of an array . A simple four-dimensional array of double-precision numbers, merely twenty elements wide in each dimension, takes up 20^4 * 8, or 1,280,000 bytes of memory - about a megabyte.

For exmaple, you have ten rows and ten columns, for a total of 100 elements. It's really no big deal. The first number in brackets is the number of rows, the second number in brackets is the number of columns. So, the upper left corner of any grid would be element [0][0]. The element to its right would be [0][1], and so on. Here is a little illustration to help.

[0][0]	[0][1]	[0][2]
[1][0]	[1][1]	[1][2]
[2][0]	[2][1]	[2][2]

Three-dimensional arrays (and higher) are stored in the same way as the two-dimensional ones. They are kept in computer memory as a linear sequence of variables, and the last index is always the one that varies fastest (then the next-to-last, and so on).

6.6.1 Multi - Dimensional Array Declaration

You can declare an array of two dimensions as follows:

datatype array_name[*size1*][*size2*];

In the above example, *variable_type* is the name of some type of variable, such as int. Also, *size1* and *size2* are the sizes of the array's first and second dimensions, respectively. Here is an example of defining an 8-by-8 array of integers, similar to a chessboard. Remember, because C arrays are zero-based, the indices on each side of the chessboard array run 0 through 7, rather than 1 through 8. The effect is the same: a two-dimensional array of 64 elements.

```
int chessboard [8][8];
```

To pinpoint an element in this grid, simply supply the indices in both dimensions.

6.6.2 Initialisation of Two - Dimensional Arrays

If you have an *m* x *n* array, it will have *m* * *n* elements and will require *m*n*element-size* bytes of storage. To allocate storage for an array you must reserve this amount of memory. The elements of a two-dimensional array are stored row wise. If table is declared as:

```
int table  [ 2 ] [ 3 ] = { 1,2,3,4,5,6 };
```

It means that element
table [0][0] = 1;
table [0][1] = 2;
table [0][2] = 3;
table [1][0] = 4;
table [1][1] = 5;
table [1][2] = 6;

The neutral order in which the initial values are assigned can be altered by including the groups in { } inside main enclosing brackets, like the following initialization as above:

```
int        table  [ 2 ] [ 3 ] = { {1,2,3},
                                  {4,5,6}    };
```

The value within innermost braces will be assigned to those array elements whose last subscript changes most rapidly. If there are few remaining values in the row, they will be assigned zeros. The number of values cannot exceed the defined row size.

```
int     table [ 2 ] [ 3 ] = { { 1, 2, 3},{ 4}};
```

It assigns values as
table [0][0] = 1;
table [0][1] = 2;
table [0][2] = 3;
table [1][0] = 4;
table [1][1] = 0;
table [1][2] = 0

Remember that, C language performs no error checking on array bounds. If you define an array with 50 elements and you attempt to access element 50 (the 51st element), or any out of bounds index, the compiler issues no warnings. It is the programmer's task to check that all attempts to access or write to arrays are done only at valid array indexes. Writing or reading past the end of arrays is a common programming bug and is hard to isolate.

Check Your Progress 3

1. Declare a multi-dimensioned array of floats called balances having three rows and five columns.
   ………………………………………………………………………………………
   ………………………………………………………………………………………
2. Write a *for* loop to total the contents of the multi-dimensioned float array balances.
   ………………………………………………………………………………………
   ………………………………………………………………………………………
3. Write a for loop which will read five characters (use scanf) and deposit them into the character based array words, beginning at element 0.
   ………………………………………………………………………………………
   ………………………………………………………………………………………

6.7 SUMMARY

Like other languages, C uses arrays as a way of describing a collection of variables with identical properties. The group has a single name for all its members, with the individual member being selected by an *index*. We have learnt in this unit, the basic purpose of using an array in the program, declaration of array and assigning values to the arrays. All elements of the arrays are stored in the consecutive memory locations**.** Without exception, all arrays in C are indexed from 0 up to one less than the bound given in the declaration. This is very puzzling for a beginner. Watch out for it in the examples provided in this unit. One important point about array declarations is that they don't permit the use of varying subscripts. The numbers given must be constant expressions which can be evaluated at compile time, not run time. As with other variables, global and static array elements are initialized to 0 by default, and automatic array elements are filled with garbage values. In C, an array of type char is used to represent a character string, the end of which is marked by a byte set to 0 (also known as a NULL character).

Whenever the arrays are passed to function their starting address is used to access rest of the elements. This is called – Call by reference. Whatever changes are made to the elements of an array in the function, they are also made available in the calling part. The formal argument contains no size specification except for the rightmost dimension. Arrays and pointers are closely linked in C. Multi-dimensional arrays are simply arrays of arrays. To use arrays effectively it is a good idea to know how to use pointers with them. More about the pointers can be learnt from Unit -10 (Block -3).

6.8 SOLUTIONS / ANSWERS

Check Your Progress 1

1. If you use a subscript that is out of bounds of the array declaration, the program will probably compile and even run. However, the results of such a mistake can be unpredictable. This can be a difficult error to find once it starts causing problems. So, make sure you're careful when initializing and accessing the array elements.

2.
   a) 6
   b) 5
   c) 5

3. This mistake doesn't produce a compiler error. If you don't initialize an array, there can be any value in the array elements. You might get unpredictable results. You should always initialize the variables and the arrays so that you know their content.

4. Each element of an array must be initialized. The safest way for a beginner is to initialize an array, either with a declaration, as shown in this chapter, or with a ***for*** statement. There are other ways to initialize an array, but they are beyond the scope of this Unit.

5. Use a ***for*** loop to total the contents of an integer array which has five elements. Store the result in an integer called total.

```
for ( loop = 0, total = 0; loop < 5; loop++ )
total = total + numbers[loop];
```

Check Your Progress 2

1. Call by reference.

2. It is possible to pass the whole array to a function. In this case, only the address of the array will be passed. When this happens, the function can change the value of the elements in the array.

3. Two common statistical applications that use arrays are:

   - **Frequency distributions**: A frequency array show the number of elements with an identical value found in a series of numbers. For example, suppose we have taken a sample of 50 values ranging from 0 to 10. We want to know how many of the values are 0, how many are 1, how many are 2 and so forth up to 10. Using the arrays we can solve the problem easily . Histogram is a pictorial representation of the frequency array. Instead of printing the values of the elements to show the frequency of each number, we print a histogram in the form of a bar chart.
   - **Random Number Permutations**: It is a set of random numbers in which no numbers are repeated. For example, given a random number permutation of 5 numbers, the values of 0 to 5 would all be included with no duplicates.

Check Your Progress 3

1.
```
float balances[3][5];
```

2.
```
for(row = 0, total = 0; row < 3; row++)
   for(column = 0; column < 5; column++)
   total = total + balances[row][column];
```

3.
```
for(loop = 0; loop < 5; loop++)
   scanf ("%c", &words[loop] );
```

6.9 FURTHER READINGS

1. The C Programming Language, *Brain W. Kernighan, Dennis M. Ritchie*, PHI.
2. C, The Complete Reference, Fourth Edition, *Herbert Schildt*, TMGH, 2002.
3. Computer Science – A Structured Programming Approach Using C, *Behrouz A. Forouzan, Richard F. Gilberg*, Thomas Learning, Second edition, 2001.
4. Programming with ANSI and TURBO C, *Ashok N. Kamthane*, Pearson Education, 2002.

UNIT 7 STRINGS

Structure

7.0 INTRODUCTION

In the previous unit, we have discussed numeric arrays, a powerful data storage method that lets you group a number of same-type data items under the same group name. Individual items, or elements, in an array are identified using a subscript after the array name. Computer programming tasks that involve repetitive data processing lend themselves to array storage. Like non-array variables, arrays must be declared before they can be used. Optionally, array elements can be initialized when the array is declared. In the earlier unit, we had just known the concept of *character arrays* which are also called *strings*.

String can be represented as a single-dimensional character type array. C language does not provide the intrinsic string types. Some problems require that the characters within a string be processed individually. However, there are many problems which require that strings be processed as complete entities. Such problems can be manipulated considerably through the use of special string oriented library functions. Most of the C compilers include string library functions that allow string comparison, string copy, concatenation of strings etc. The string functions operate on null-terminated arrays of characters and require the header <string.h>.The use of the some of the string library functions are given as examples in this unit.

7.1 OBJECTIVES

After going through this unit, you will be able to:

- define, declare and initialize a string;
- discuss various formatting techniques to display the strings; and
- discuss various built-in string functions and their use in manipulation of strings.

7.2 DECLARATION AND INITIALIZATION OF STRINGS

Strings in C are group of characters, digits, and symbols enclosed in quotation marks or simply we can say the string is declared as a “character array”. The end of the string is marked with a special character, the ‘\0’ (*Null character)*, which has the decimal value 0. There is a difference between a *character* stored in memory and a *single character string* stored in a memory. The character requires only one byte whereas the single character string requires two bytes (one byte for the character and other byte for the delimiter).

Declaration of strings

A string in C is simply a sequence of characters. To declare a string, specify the data type as char and place the number of characters in the array in square brackets after the string name. The syntax is shown as below:

char string-name[size];

For example,

```
char name[20];
char address[25];
char city[15];
```

Initialization of strings

The string can be initialized as follows:

char name[8] = {'P', 'R', 'O', 'G', 'R', 'A', 'M', '\0'};

Each character of string occupies 1 byte of memory (on 16 bit computing). The size of character is machine dependent, and varies from 16 bit computers to 64 bit computers. The characters of strings are stored in the contiguous (adjacent) memory locations.

1 byte	1 byte	1 byte	1 byte	1 byte	1 byte	1 byte	1 byte
P	R	O	G	R	A	M	\0
1001	1002	1003	1004	1005	1006	1007	1008

The C compiler inserts the NULL (\0) character automatically at the end of the string. So initialization of the NULL character is not essential.

You can set the initial value of a character array when you declare it by specifying a string literal. If the array is too small for the literal, the literal will be truncated. If the literal (including its null terminator) is smaller than the array, then the final characters in the array will be undefined. If you don't specify the size of the array, but do specify a literal, then C will set the array to the size of the literal, including the null terminator.

```
  char str[4] = {'u', 'n', 'i', 'x'};
  char str[5] = {'u', 'n', 'i', 'x', '\0'};
  char str[3];
  char str[ ] =  "UNIX";
  char str[4] = "unix";
  char str[9] = "unix";
```

All of the above declarations are legal. But which ones don't work? The first one is a valid declaration, but will cause major problems because it is not *null-terminated.* The second example shows a correct null-terminated string. The special escape character **\0** denotes string termination. The fifth example suffers the size problem, the character array ***'str'*** is of size 4 bytes, but it requires an additional space to store ***'\0'***. The fourth example however does not. This is because the compiler will determine the length of the string and automatically initialize the last character to a null-terminator. The strings not terminated by a ***'\0'*** are merely a collection of characters and are called as *character arrays*.

String Constants

String constants have double quote marks around them, and can be assigned to char pointers. Alternatively, you can assign a string constant to a char array - either with no size specified, or you can specify a size, but don't forget to leave a space for the null character! Suppose you create the following two code fragments and run them:

```
/*   Fragment 1   */
{
   char *s;
   s=hello";
   printf("%s\n",s);
}

/*  Fragment  2 */

{
   char s[100];
   strcpy(s, " hello");
   printf("%s\n",s);
}
```

These two fragments produce the same output, but their internal behaviour is quite different. In fragment 2, you cannot say **s = "hello";**. To understand the differences, you have to understand how the *string constant table* works in C. When your program is compiled, the compiler forms the object code file, which contains your machine code and a table of all the string constants declared in the program. In fragment 1, the statement **s = "hello";** causes ***s*** to point to the address of the string **hello** in the string constant table. Since this string is in the string constant table, and therefore technically a part of the executable code, you cannot modify it. You can only point to it and use it in a read-only manner. In fragment 2, the string **hello** also exists in the constant table, so you can copy it into the array of characters named ***s***. Since ***s*** is not an address, the statement **s="hello";** will not work in fragment 2. It will not even compile.

Example 7.1

Write a program to read a name from the keyboard and display message **Hello** onto the monitor

Program 7.1

```
/*Program that reads the name and display the hello along with your name*/
#include <stdio.h>
main()
{
char name[10];
printf("\nEnter Your Name : ");
scanf("%s", name);
printf("Hello %s\n", name);
}
```

OUTPUT

```
Enter Your Name :  Alex
Hello Alex
```

In the above example declaration char name [10] allocates 10 bytes of memory space (on 16 bit computing) to array name [ ]. We are passing the base address to scanf function and scanf() function fills the characters typed at the keyboard into array until enter is pressed. The scanf() places '\0' into array at the end of the input. The printf()

function prints the characters from the array on to monitor, leaving the end of the string '\0'. The *%s* used in the scanf() and printf() functions is a format specification for strings.

7.3 DISPLAY OF STRINGS USING DIFFERENT FORMATTING TECHNIQUES

The ***printf*** function with ***%s*** format is used to display the strings on the screen. For example, the below statement displays entire string:

printf("%s", name);

We can also specify the accuracy with which character array (string) is displayed. For example, if you want to display first 5 characters from a field width of 15 characters, you have to write as:

printf("%15.5s", name);

If you include minus sign in the format (e.g. % –10.5s), the string will be printed left justified.

printf("% -10.5s", name);

Example 7.2

Write a program to display the string "UNIX" in the following format.

```
U
UN
UNI
UNIX
UNIX
UNI
UN
U
```

```
/* Program to display the string in the above shown format*/

# include <stdio.h>
main()
{
int  x, y;
static char string[ ] = "UNIX";
printf("\n");
for( x=0; x<4; x++)
{
        y = x + 1;
   /* reserves 4 character of space on to the monitor and minus sign is for left
justified*/
        printf("%-4.*s \n", y, string);

  /* and for every loop the * is replaced by value of y */
/* y value starts with 1 and for every time it is incremented by 1 until it reaches to 4*/
}

for( x=3; x>=0; x- -)
        {
         y = x + 1;
         printf("%-4.*s \n", y, string);
/*  y value starts with 4 and for every time it is decrements by 1 until it reaches to 1*/
        }
}
```

OUTPUT

```
U
UN
UNI
UNIX
UNIX
UNI
UN
U
```

7.4 ARRAY OF STRINGS

Array of strings are multiple strings, stored in the form of table. Declaring array of strings is same as strings, except it will have additional dimension to store the number of strings. Syntax is as follows:

char array-name[size][size];

For example,

char names[5][10];

where names is the name of the character array and the constant in first square brackets will gives number of string we are going to store, and the value in second square bracket will gives the maximum length of the string.

Example 7.3

char names [3][10] = {“martin”, “phil”, “collins”};

It can be represented by a two-dimensional array of size[3][10] as shown below:

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
m	a	r	t	i	n	\0			
p	h	i	l	\0					
c	o	l	l	i	n	s	\0		

Example 7.4

Write a program to initializes 3 names in an array of strings and display them on to monitor

```
/* Program that initializes 3 names in an array of strings and display them on to
monitor.*/

#include <stdio.h>
main()
{
        int   n;
        char   names[3][10] = {“Alex”, “Phillip”, “Collins” };
        for(n=0; n<3; n++)
        printf(“%s \n”,names[n] );    }
```

OUTPUT

```
Alex
Phillip
Collins
```

Check Your Progress 1

1. Which of the following is a static string?

   A. Static String;
   B. “Static String”;
   C. ‘Static String’;
   D. char string[100];

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

2. Which character ends all strings?

   A. ‘.’
   B. ‘ ‘
   C. ‘0’
   D. ‘n’

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

3. What is the Output of the following programs?

(a)
```
main()
{
     char name[10] = “IGNOU”;
     printf(“\n %c”, name[0]);
     printf(“\n %s”, name);
}
```

(b)
```
main()
{
  char  s[ ] = “hello”;
  int j = 0;
  while ( s[j] != ‘\0’ )
       printf(“ %c”,s[j++]);
}
```

(c)
```
main()
{
 char  str[ ] = “hello”;
 printf(“%10.2s”, str);
 printf(“%-10.2s”, str);
}
```

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

4 Write a program to read 'n' number of lines from the keyboard using a two-dimensional character array (ie., strings).

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

7.5 BUILT IN STRING FUNCTIONS AND APPLICATIONS

The header file <string.h> contains some string manipulation functions. The following is a list of the common string managing functions in C.

7.5.1 Strlen Function

The **strlen** function returns the length of a string. It takes the string name as argument. The syntax is as follows:

n = strlen (str);

where **str** is name of the string and **n** is the length of the string, returned by **strlen** function.

Example 7. 5

Write a program to read a string from the keyboard and to display the length of the string on to the monitor by using strlen( ) function.

```
/*  Program to illustrate the strlen function to determine the length of a string */

#include <stdio.h>
#include <string.h>
main()
{
char name[80];
int length;
printf("Enter your name: ");
gets(name);
length = strlen(name);
printf("Your name has %d characters\n", length);
}
```

OUTPUT

```
Enter your name: TYRAN
Your name has 5 characters
```

7.5.2 Strcpy Function

In C, you cannot simply assign one character array to another. You have to copy element by element. The string library <string.h> contains a function called **strcpy** for this purpose. The **strcpy** function is used to copy one string to another. The syntax is as follows:

strcpy(str1, str2);

where str1, str2 are two strings. The content of string str2 is copied on to string str1**.**

Example 7.6

Write a program to read a string from the keyboard and copy the string onto the second string and display the strings on to the monitor by using strcpy( ) function.

```
/* Program to illustrate strcpy function*/

#include <stdio.h>
#include <string.h>
main()
{
char first[80], second[80];
printf("Enter a string: ");
gets(first);
strcpy(second, first);
printf("\n First string is : %s, and second string is: %s\n", first, second);
}
```

OUTPUT

```
Enter a string: ADAMS
First string is: ADAMS, and second string is: ADAMS
```

7.5.3 Strcmp Function

The **strcmp** function in the string library function which compares two strings, character by character and stops comparison when there is a difference in the ASCII value or the end of any one string and returns ASCII difference of the characters that is integer. If the return value ***zero*** means the two strings are equal, a negative value means that first is less than second, and a positive value means first is greater than second**.** The syntax is as follows:

n = strcmp(str1, str2);
where **str1** and **str2** are two strings to be compared and **n** is returned value of differed characters.

Example 7.7

Write a program to compare two strings using string compare function.

/* The following program uses the **strcmp** function to compare two strings. */

```
#include <stdio.h>
#include <string.h>
main()
{
 char first[80], second[80];
 int value;
printf("Enter a string: ");
 gets(first);
 printf("Enter another string: ");
 gets(second);
 value = strcmp(first,second);
  if(value == 0)
       puts("The two strings are equal");
     else if(value < 0)
         puts("The first string is smaller ");
        else if(value > 0)
        puts("the first string is bigger");
}
```

OUTPUT

```
Enter a string: MOND
Enter another string: MOHANT
The first string is smaller
```

7.5.4 Strcat Function

The **strcat** function is used to join one string to another. It takes two strings as arguments; the characters of the second string will be appended to the first string. The syntax is as follows:

strcat(str1, str2);
where str1 and str2 are two string arguments, string *str2* is appended to string *str1***.**

Example 7.8

Write a program to read two strings and append the second string to the first string.

```
/* Program for string concatenation*/

#include <stdio.h>
#include <string.h>
main()
{
char first[80], second[80];
printf("Enter a string:");
gets(first);
printf("Enter another string: ");
gets(second);
strcat(first, second);
printf("\nThe two strings joined together: %s\n", first);
}
```

OUTPUT

```
Enter a string: BOREX
Enter another string: BANKS
The two strings joined together: BOREX BANKS
```

7.5.5 Strlwr Function

The **strlwr** function converts upper case characters of string to lower case characters. The syntax is as follows:

strlwr(str1);
where str1 is string to be converted into lower case characters.

Example 7.9

Write a program to convert the string into lower case characters using in-built function.

```
/* Program that converts input string to lower case characters */

#include <stdio.h>
#include <string.h>
```

```
main()
{
char first[80];
printf("Enter a string: ");
gets(first);
printf("Lower case of the string is %s”, strlwr(first));
}
```

OUTPUT

```
Enter a string: BROOKES
Lower case of the string is brookes
```

7.5.6 Strrev Function

The **strrev** funtion reverses the given string. The syntax is as follows:

strrev(str);
where string **str** will be reversed.

Example 7.9

Write a program to reverse a given string.

```
/* Program to reverse a given string */

#include <stdio.h>
#include <string.h>
main()
{
char first[80];
printf(“Enter a string:”);
gets(first);
printf(“\n Reverse of the given string is :  %s ”, strrev(first));
}
```

OUTPUT

```
Enter a string: ADANY
Reverse of the given string is:  YNADA
```

7.5.7 Strspn Function

The **strspn** function returns the position of the string, where first string mismatches with second string. The syntax is as follows:

n = strspn (first, second);
where **first** and **second** are two strings to be compared, **n** is the number of character from which first string does not match with second string.

Example 7.10

Write a program, which returns the position of the string from where first string does not match with second string.

```
/*Program which returns the position of the string from where first string does not
match with second string*/

#include <stdio.h>
#include <string.h>
main()
```

```
{
char first[80], second[80];
printf("Enter  first string: “);
gets(first);
printf(“\n Enter  second string: “);
gets(second);
printf(“\n After %d characters there is no match”,strspn(first, second));
}
```

OUTPUT

```
Enter first string: ALEXANDER
Enter second string: ALEXSMITH
After 4 characters there is no match
```

7.6 OTHER STRING FUNCTIONS

strncpy function

The **strncpy** function same as *strcpy*. It copies characters of one string to another string up to the specified length. The syntax is as follows:

strncpy(str1, str2, 10);
where str1 and str2 are two strings. The **10** characters of string **str2** are copied onto string **str1.**

stricmp function

The **stricmp** function is same as *strcmp*, except it compares two strings ignoring the case (lower and upper case). The syntax is as follows:

n = stricmp(str1, str2);

strncmp function

The **strncmp** function is same as *strcmp*, except it compares two strings up to a specified length. The syntax is as follows:

n = strncmp(str1, str2, 10);
where **10** characters of **str1** and **str2** are compared and **n** is returned value of differed characters.

strchr function

The **strchr** funtion takes two arguments (the string and the character whose address is to be specified) and returns the address of first occurrence of the character in the given string. The syntax is as follows:

cp = strchr (str, c);
where **str** is string and **c** is character and **cp** is character pointer.

strset function

The **strset** funtion replaces the string with the given character**.** It takes two arguments the string and the character. The syntax is as follows:

strset (first, ch);
where string **first** will be replaced by character **ch**.

strchr function

The **strchr** function takes two arguments (the string and the character whose address is to be specified) and returns the address of first occurrence of the character in the given string. The syntax is as follows:

cp = strchr (str, c);
where **str** is string and **c** is character and **cp** is character pointer.

strncat function

The **strncat** function is the same as *strcat*, except that it appends upto specified length. The syntax is as follows:

strncat(str1, str2,10);
where 10 character of the str2 string is added into str1 string.

strupr function

The strupr function converts lower case characters of the string to upper case characters. The syntax is as follows:

strupr(str1);
where str1 is string to be converted into upper case characters.

strstr function

The **strstr** function takes two arguments address of the string and second string as inputs. And returns the address from where the second string starts in the first string. The syntax is as follows:

cp = strstr (first, second);
where **first** and s**econd** are two strings, **cp** is character pointer.

Check Your Progress 2

1. Which of the following functions compares two strings?
   A. compare();
   B. stringcompare();
   C. cmp();
   D. strcmp();

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

2. Which of the following appends one string to the end of another?
   A. append();
   B. stringadd();
   C. strcat();
   D. stradd();

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

3. Write a program to concatenate two strings without using the *strcat()* function.

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

4. Write a program to find string length without using the *strlen()* function.

………………………………………………………………………………………

………………………………………………………………………………………

………………………………………………………………………………………

5. Write a program to convert lower case letters to upper case letters in a given string without using strupp().

………………………………………………………………………………………

………………………………………………………………………………………

………………………………………………………………………………………

7.7 SUMMARY

Strings are sequence of characters. Strings are to be null-terminated if you want to use them properly. Remember to take into account null-terminators when using dynamic memory allocation. The string.h library has many useful functions. Losing the ‘ **\0’** character can lead to some very considerable bugs. Make sure you copy \0 when you copy strings. If you create a new string, make sure you put \0 in it. And if you copy one string to another, make sure the receiving string is big enough to hold the source string, including \0. Finally, if you point a character pointer to some characters, make sure they end with \0.

String Functions	**Its Use**
strlen	Returns number of characters in string.
strlwr	Converts all the characters in the string into lower case characters
strcat	Adds one string at the end of another string
strcpy	Copies a string into another
strcmp	Compares two strings and returns zero if both are equal.
strdup	Duplicates a string
strchr	Finds the first occurrence of given character in a string
strstr	Finds the first occurrence of given string in another string
strset	Sets all the characters of string to given character or symbol
strrev	Reverse a string

7.8 SOLUTIONS / ANSWERS

Check Your Progress 1

1. B

2. C

3. (a) I
   IGNOU

   (b) hello

   (c) hehe

Check Your Progress 2

1. D

2. C

3. /* Program to concatenate two strings without using the strcat() function*/

```
# include<string.h>
# include <stdio.h>
 main()
 {
  char str1[10];
  char str2[10];
  char output_str[20];
  int i=0, j=0, k=0;
 printf(" Input the first string: ");
            gets(str1);
            printf("\nInput the second string: ");
            gets(str2);
          while(str1[i] != '\0')
  output_str[k++] = str1[i++];
 while(str2[j] != '\0')
  output_str[k++] = str2[j++];
 output_str[k] = '\0';
          puts(output_str);
 }
```

4. /* Program to find the string length without using the strlen() funtion */

```
# include<stdio.h>
# include<string.h>
main()
{
char  string[60];
int len=0, i=0;
printf(" Input the string : ");
gets(string);
while(string[i++] != '\0')
         len ++;
printf("Length of Input String = %d", len);
getchar();
}
```

5. /* Program to convert the lower case letters to upper case in a given string without using strupp() function*/

```
#include<stdio.h>
main()
{
int i= 0;   char source[10], destination[10];
gets(source);
while( source[i] != '\0')
{
  if((source[i]>=97) && (source[i]<=122))
```

```
                destination[i]=source[i]-32;
            else
                 destination[i]=source[i];
   i++;
   }
   destination[i]= ‘ \0 ’;
   puts(destination);
 }
```

7.9 FURTHER READINGS

1. The C programming language, *Brain W. Kernighan, Dennis M. Ritchie*, PHI.
2. Programming with ANSI and Turbo C, *Ashok N. Kamthane*, Pearson Education, 2002.
3. Computer Programming in C, *Raja Raman. V*, 2002, PHI.
4. C,The Complete Reference, Fourth Edition, *Herbert Schildt*, Tata McGraw Hill, 2002.
5. Computer Science A structured Programming Approach Using C, *Behrouz A. Forouzan, Richard F. Gilberg*, Brooks/Cole Thomas Learning, Second Edition, 2001.

UNIT 8 FUNCTIONS

Structure

8.0 INTRODUCTION

To make programming simple and easy to debug, we break a larger program into smaller *subprograms* which perform '*well defined tasks*'. These subprograms are called *functions.* So far we have defined a single function *main ( ).*

After reading this unit you will be able to define many other functions and the *main( )* function can call up these functions from several different places within the program, to carry out the required processing.

Functions are very important tools for ***Modular Programming***, where we break large programs into small subprograms or modules (functions in case of C). The use of functions reduces complexity and makes programming simple and easy to understand.

In this unit, we will discuss how functions are defined and how are they accessed from the main program? We will also discuss various types of functions and how to invoke them. And finally you will learn an interesting and important programming technique known as *Recursion*, in which a function calls within itself.

8.1 OBJECTIVES

After going through this unit, you will learn:

- the need of functions in the programming;
- how to define and declare functions in 'C' Language;
- different types of functions and their purpose;
- how the functions are called from other functions;
- how data is transferred through parameter passing, to functions and the Return statement;
- recursive functions; and
- the concept of 'C*all by Value*' and its drawbacks.

8.2 DEFINTION OF A FUNCTION

A ***function*** is a self- contained block of executable code that can be called from any other function .In many programs, a set of statements are to be executed repeatedly at various places in the program and may with different sets of data, the idea of functions comes in mind. You keep those repeating statements in a function and call them as and when required. When a function is called, the control transfers to the called function, which will be executed, and then transfers the control back to the calling function (to the statement following the function call). Let us see an example as shown below:

Example 8.1

```
/* Program to illustrate a function*/

#include <stdio.h>
main ()
{
void sample( );
printf("\n You are in main");
}

void sample( )
 {
 printf("\n You are in sample");
  }
```

OUTPUT

```
You are in sample
You are in main
```

Here we are calling a function ***sample ( )*** through ***main( )*** i.e. control of execution transfers from ***main( )*** to ***sample( )*** , which means ***main( )*** is suspended for some time and ***sample( )*** is executed. After its execution the control returns back to ***main( )***, at the statement following function call and the execution of ***main( )*** is resumed.

The syntax of a function is:

```
return  data type  function_name (list of arguments)
{
        datatype declaration of the arguments;
        executable statements;
        return (expression);
}
```

where,

- return data type is the same as the data type of the variable that is returned by the function using return statement.
- a function_name is formed in the same way as variable names / identifiers are formed.
- the list of arguments or parameters are valid variable names as shown below, separated by commas: (data type1 var1,data type2 var2,…….. data type n var n) for example *(int x, float y, char z).*
- arguments give the values which are passed from the calling function.

- the body of function contains executable statements.
- the return statement returns a ***single*** value to the calling function.

Example 8.2

Let us write a simple function that calculates the square of an integer.

```
/*Program to calculate the square of a given integer*/

/* square( ) function */
{
  int square (int no)                /*passing of argument */
 int result ;              /* local variable to function square */
 result = no*no;
 return (result);                    /* returns an integer value */
          }

 /*It will be called from main()as follows */
main( )
{
int n ,sq;                 /* local variable to function main */
printf ("Enter a number to calculate square value");
scanf("%d",&n);
sq=square(n);              /* function call with parameter passing */
printf ("\nSquare of  the number is : %d", sq);
}  /* program  ends */
```

OUTPUT

```
Enter a number to calculate square value : 5
Square of the number is : 25
```

8.3 DECLARATION OF A FUNCTION

As we have mentioned in the previous section, every function has its declaration and function definition. When we talk of declaration only, it means only the function name, its argument list and return type are specified and the function body or definition is not attached to it. The *syntax* of a function declaration is:

return data type function_name(list of arguments);

For example,

```
int square(int no);
float temperature(float c, float f);
```

We will discuss the use of function declaration in the next section.

8.4 FUNCTION PROTOTYPES

In Example 8.1 for calculating square of a given number, we have declared function ***square( )*** before ***main( )*** function; this means before coming to ***main( )***, the compiler knows about ***square( )***, as the compilation process starts with the first statement of any program. Now suppose, we reverse the sequence of functions in this program i.e., writing the ***main( )*** *function* and later on writing the ***square( )*** function, *what happens* ? The "C" compiler will give an error. Here the introduction of concept of *"function prototypes"* solves the above problem.

Function Prototypes require that every function which is to be accessed should be declared in the calling function. The function declaration, that will be discussed earlier, will be included for every function in its calling function . Example 8.2 may be modified using the function prototype as follows:

Example 8.3

```
/*Program to calculate the square of a given integer using the function prototype*/
#include <stdio.h>
main ( )
{
int n , sq ;
int square (int ) ;                    /* function prototype */
printf ("Enter a number to calculate square value");
scanf("%d",&n);
sq = square(n);              /* function call with parameter passing */
printf ("\nSsquare of  the number is : %d", sq);
}

/* square function */
int square (int no)                    /*passing of argument */
{
  int result ;               /* local variable to function square */
  result = no*no;
  return (result);                     /* returns an integer value */
}
```

OUTPUT

```
Enter a number to calculate square value :  5
Square of the number is:  25
```

Points to remember:

- *Function prototype* requires that the function declaration must include the return type of function as well as the type and number of arguments or parameters passed.
- The variable names of arguments need not be declared in prototype.
- The major reason to use this concept is that they enable the compiler to check if there is any mismatch between function declaration and function call.

Check Your Progress 1

(1) Write a function to multiply two integers and display the product.

……………………………………………………………………………………

……………………………………………………………………………………

(2) Modify the above program, by introducing function prototype in the main function.

……………………………………………………………………………………

……………………………………………………………………………………

8.5 THE *return* STATEMENT

If a function has to return a value to the calling function, it is done through the ***return*** statement. It may be possible that a function does not return any value; only the control is transferred to the calling function. The syntax for the *return* statement is:

return (expression);

We have seen in the *square( )* function, the *return* statement, which returns an integer value.

Points to remember:

- You can pass any number of arguments to a function but can return only one value at a time.

  For example, the following are the valid *return* statements

  (a) return (5);
  (b) return (x*y);

  For example, the following are the invalid *return* statements
  (c) return (2, 3);
  (d) return (x, y);

- If a function does not return anything, ***void*** specifier is used in the function declaration.

  For example:

```
void square (int no)
 {
 int sq;
 sq = no*no;
 printf ("square is %d", sq);
 }
```

- All the function's return type is by default is ***"int"***, i.e. a function returns an integer value, if no type specifier is used in the function declaration.

  Some examples are:

```
(i)   square (int no);            /* will return an integer value */

(ii)  int square (int no);        /* will return an integer value */

(iii) void square (int no);       /* will not return anything  */
```

- What happens if a function has to return some value other than integer? The answer is very simple: use the particular type specifier in the function declaration.

For example consider the code fragments of function definitions below:

1) **Code Fragment - 1**

```
char func_char( …….. )
{
      char c;
…………….
…………….
……………
}
```

2) **Code Fragment - 2**

```
float func_float (……..)
{
  float f;
      …………..
      …………..
      …………..
      return(f);
 }
```

Thus from the above examples, we see that you can return all the data types from a function, the only condition being that the value returned using return statement and the type specifier used in function declaration should match.

- A function can have many *return* statements. This thing happens when some condition based returns are required.

  For example,

```
/*Function to find greater of two numbers*/
int greater (int x, int y)
  {
      if (x>y)
          return (x);
       else
          return (y);
    }
```

- And finally, with the execution of return statement, the control is transferred to the calling function with the value associated with it.

  In the above example if we take x = 5 and y = 3, then the control will be transferred to the calling function when the first return statement will be encountered, as the condition (x > y) will be satisfied. All the remaining executable statements in the function will not be executed after this returning.

Check Your Progress 2

1. Which of the following are valid return statements?

   a) return (a);
   b) return (z,13);
   c) return (22.44);
   d) return;
   e) return (x*x, y*y);

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

8.6 TYPES OF VARIABLES AND STORAGE CLASSES

In a program consisting of a number of functions a number of different types of variables can be found.

Global vs. Static variables: Global variables are recognized through out the program whereas local valuables are recognized only within the function where they are defined.

Static vs. Dynamic variables: Retention of value by a local variable means, that in static, retention of the variable value is lost once the function is completely executed whereas in certain conditions the value of the variable has to be retained from the earlier execution and the execution retained.

The variables can be characterized by their ***data type*** and by their ***storage class***. One way to classify a variable is according to its data type and the other can be through its storage class. ***Data type*** refers to the type of value represented by a variable whereas ***storage*** class refers to the ***permanence*** of a variable and its scope within the program i.e. portion of the program over which variable is recognized.

Storage Classes

There are four different storage classes specified in C:

1.	Auto (matic)	2.	Extern (al)
3.	Static	4.	Register

The storage class associated with a variable can sometimes be established by the location of the variable declaration within the program or by prefixing keywords to variables declarations.

```
For example:  auto    int     a, b;
              static  int     a, b;
              extern  float   f;
```

8.6.1 Automatic Variables

The variables local to a function are automatic i.e., declared within the function. The scope of lies within the function itself. The automatic defined in different functions, even if they have same name, are treated as different. It is the default storage class for variables declared in a function.

Points to remember:

- The auto is optional therefore there is no need to write it.
- All the formal arguments also have the auto storage class.
- The initialization of the auto-variables can be done:
    - in declarations
    - using assignment expression in a function
- If not initialized the unpredictable value is defined.
- The value is not retained after exit from the program.

Let us study these variables by a sample program given below:

Example 8.4

```
/* To print the value of automatic variables */

# include <stdio.h>
main ( int argc, char * argv[  ])
{
int    a, b;
double  d;
printf("%d",   argc);
a = 10;
b = 5;
d = (b * b) – (a/2);
printf("%d, %d, %f", a, b, d);
}
```

All the variables a, b, d, argc and argv [ ] have automatic storage class.

8.6.2 External (Global) Variables

These are not confined to a single function. Their scope ranges from the point of declaration to the entire remaining program. Therefore, their scope may be the entire program or two or more functions depending upon where they are declared.

Points to remember:

- These are global and can be accessed by any function within its scope. Therefore value may be assigned in one and can be written in another.
- There is difference in external variable definition and declaration.
- External Definition is the same as any variable declaration:
    - Usually lies outside or before the function accessing it.
- It allocates storage space required.
- Initial values can be assigned.
- The external specifier is not required in external variable definition.
- A declaration is required if the external variable definition comes after the function definition.
- A declaration begins with an external specifier.
- Only when external variable is defined is the storage space allocated.
- External variables can be assigned initial values as a part of variable definitions, but the values must be constants rather than expressions.
- If initial value is not included then it is automatically assigned a value of zero.

Let us study these variables by a sample program given below:

Example 8.5

```
/* Program to illustrate the use of global variables*/

#  include <stdio.h>
int gv;                                    /*global variable*/
main  (  )
{
void function1();                /*function declaration*/
gv = 10;
printf  ("%d is the value of gv before function call\n", gv);
function1( );
printf  ("%d is the value of gv after function call\n", gv);
}
```

```
void function1 ( )
{
gv = 15: }
```

OUTPUT

```
10 is the value of gv before function call
15 is the value of gv after function call
```

8.6.3 Static Variables

In case of single file programs static variables are defined within functions and individually have the same scope as automatic variables. But static variables retain their values throughout the execution of program within their previous values.

Points to remember:

- The specifier precedes the declaration. Static and the value cannot be accessed outside of their defining function.
- The static variables may have same name as that of external variables but the local variables take precedence in the function. Therefore external variables maintain their independence with locally defined auto and static variables.
- Initial value is expressed as the constant and not expression.
- Zeros are assigned to all variables whose declarations do not include explicit initial values. Hence they always have assigned values.
- Initialization is done only is the first execution.

Let us study this sample program to print value of a static variable:

Example 8.6

```
/* Program to illustrate the use of static variable*/

#include <stdio.h>

main()
{
int call_static();
int  i,j;
i=j=0;
j = call_static();
printf("%d\n",j);
j = call_static ();
printf("%d\n",j);
j = call_static();
printf("%d\n",j);
}

int    call_static()
{
static  int  i=1;
int   j;
j  =  i;
i++;
return(j);
}
```

OUTPUT

```
1
2
3
```

This is because ***i*** is a static variable and retains its previous value in next execution of function call_static( ). To remind you ***j*** is having auto storage class. Both functions main and call_static have the same local variable ***i*** and ***j*** but their values never get mixed.

8.6.4 Register Variables

Besides three storage class specifications namely, Automatic, External and Static, there is a *register* storage class. *Registers* are special storage areas within a computer's CPU. All the arithmetic and logical operations are carried out with these registers.

For the same program, the execution time can be reduced if certain values can be stored in registers rather than memory. These programs are smaller in size (as few instructions are required) and few data transfers are required. The reduction is there in machine code and not in source code. They are declared by the proceeding declaration by register reserved word as follows:

register int m;

Points to remember:

- These variables are stored in registers of computers. If the registers are not available they are put in memory.
- Usually 2 or 3 register variables are there in the program.
- Scope is same as automatic variable, local to a function in which they are declared.
- Address operator '&' cannot be applied to a register variable.
- If the register is not available the variable is though to be like the automatic variable.
- Usually associated integer variable but with other types it is allowed having same size (short or unsigned).
- Can be formal arguments in functions.
- Pointers to register variables are not allowed.
- These variables can be used for loop indices also to increase efficiency.

8.7 TYPES OF FUNCTION INVOKING

We categorize a function's invoking (calling) depending on arguments or parameters and their returning a value. In simple words we can divide a function's invoking into four types depending on whether parameters are passed to a function or not and whether a function returns some value or not.

The various types of invoking functions are:

- With no arguments and with no return value.
- With no arguments and with return value
- With arguments and with no return value
- With arguments and with return value.

Let us discuss each category with some examples:

TYPE 1: With no arguments and have no return value

As the name suggests, any function which ***has no arguments and does not return any values to the calling function***, falls in this category. These type of functions are confined to themselves i.e. neither do they receive any data from the calling function nor do they transfer any data to the calling function. So there is no data communication between the calling and the called function are only program control will be transferred.

Example 8.7

```
/* Program for illustration of the function with no arguments and no return value*/

/* Function with no arguments and no return value*/

#include <stdio.h>
main()
 {
void message();
printf(“Control is in main\n”);
message();                    /* Type 1 Function */
printf(“Control is again in main\n”);
 }

void message()
{
  printf(“Control is in message function\n”);
  }                           /* does not return anything */
```

OUTPUT

```
Control is in main
Control is in message function
Control is again in main
```

TYPE 2: With no arguments and with return value

Suppose if a function does not receive any data from calling function but does send some value to the calling function, then it falls in this category.

Example 8.8

Write a program to find the sum of the first ten natural numbers.

```
/* Program to find sum of first ten natural numbers */

#include <stdio.h>

int cal_sum()
 {
int i, s=0;
for (i=0; i<=10; i++)
s=s + i;
return(s);                    /* function returning sum of first ten natural numbers */
}

main()
 {
int sum;
sum = cal_sum();
printf("Sum of first ten natural numbers is % d\n", sum);
}
```

OUTPUT

```
Sum of first ten natural numbers is 55
```

TYPE 3: With Arguments and have no return value

If a function ***includes arguments but does not return anything***, it falls in this category. One way communication takes place between the calling and the called function.

Before proceeding further, first we discuss the *type of arguments or parameters* here. There are two types of arguments:

- Actual arguments
- Formal arguments

Let us take an example to make this concept clear:

Example 8.9

Write a program to calculate sum of any three given numbers.

```
#include <stdio.h>

main()
{
int a1, a2, a3;
void sum(int, int, int);
printf("Enter three numbers: ");
scanf ("%d%d%d",&a1,&a2,&a3);
sum (a1,a2,a3);    /*  Type 3 function */
}


/* function to calculate sum of three numbers */
void sum (int f1, int f2, int f3)
{
int s;
 s = f1+ f2+ f3;
printf ("\nThe sum of the three numbers is %d\n",s);
}
```

OUTPUT

```
Enter three numbers: 23 34 45
The sum of the three numbers is 102
```

Here f1, f2, f3 are ***formal arguments*** and a1, a2, a3 are ***actual arguments***.
Thus we see in the function declaration, the arguments are formal arguments, but when values are passed to the function during function call, they are actual arguments.

Note: The actual and formal arguments should match in type, order and number

TYPE 4: With arguments function and with return value

In this category two-way communication takes place between the calling and called function i.e. a function returns a value and also arguments are passed to it. We modify above Example according to this category.

Example 8.10

Write a program to calculate sum of three numbers.

```
/*Program to calculate the sum of three numbers*/

#include <stdio.h>
main ( )
 {
int a1, a2, a3, result;
int sum(int, int, int);
printf("Please enter any 3 numbers:\n");
scanf ("%d %d %d", & a1, &a2, &a3);
result = sum (a1,a2,a3);   /* function call */
printf ("Sum of the given numbers is : %d\n", result);
}

/* Function to calculate the sum of three numbers */
int sum (int f1, int f2, int f3)
 {
  return(f1+ f2 + f3);  /* function returns a value */
 }
```

OUTPUT

```
Please enter any 3 numbers:
3 4 5
Sum of the given numbers is: 12
```

8.8 CALL BY VALUE

So far we have seen many functions and also passed arguments to them, but if we observe carefully, we will see that we have always created new variables for arguments in the function and then passed the values of actual arguments to them. Such function calls are called ***"call by value"***.

Let us illustrate the above concept in more detail by taking a simple function of multiplying two numbers:

Example 8.11

Write a program to multiply the two given numbers

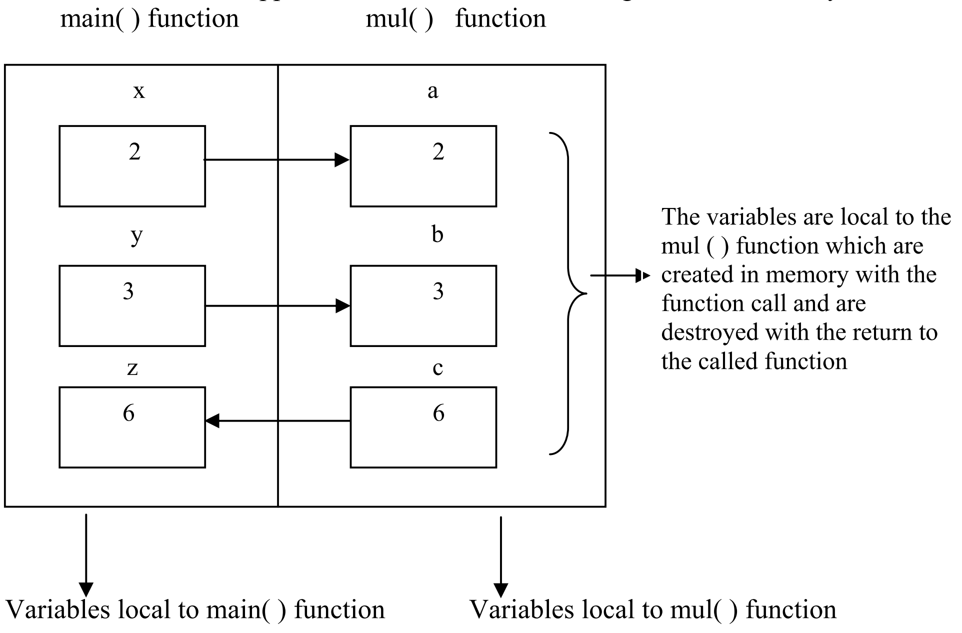
```
#include <stdio.h>
main()
{
int x, y, z;
int mul(int, int);
printf ("Enter two numbers: \n");
scanf ("%d %d",&x,&y);
z= mul(x, y);                /* function call by value */
printf ("\n The product of the two numbers is : %d", z);
}
```

```
/* Function to multiply two numbers */
int mul(int a, int b)
{
int c;
c =a*b;
return(c);    }
```

OUTPUT

```
Enter two numbers:
23  2
The product of two numbers is: 46
```

Now let us see what happens to the actual and formal arguments in memory.


main( ) function
mul( ) function
x
2
y
3
z
6
a
2
b
3
c
6
The variables are local to the mul ( ) function which are created in memory with the function call and are destroyed with the return to the called function
Variables local to main( ) function
Variables local to mul( ) function

What are meant by local variables? The answer is *local variables are those which can be used only by that function.*

Advantages of Call by value:

The only advantage is that this mechanism is simple and it reduces confusion and complexity.

Disadvantages of Call by value:

As you have seen in the above example, there is separate memory allocation for each of the variable, so unnecessary utilization of memory takes place.

The second disadvantage, which is very important from programming point of view, is that any changes made in the arguments are not reflected to the calling function, as these arguments are local to the called function and are destroyed with function return.

Let us discuss the second disadvantage more clearly using one example:

Example 8.12

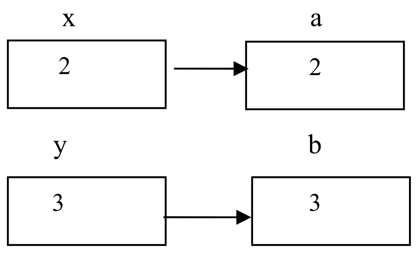
Write a program to swap two values.

```
/*Program to swap two values*/

#include <stdio.h>
main ( )
{
int x = 2, y = 3;
void swap(int, int);

printf ("\n Values before swapping are %d %d", x, y);
swap (x, y);
printf ("\n Values after swapping are %d %d", x, y);
}

/* Function to swap(interchange) two values */
void swap( int a, int b )
{
int t;
t = a;
a = b;
b = t;
}
```

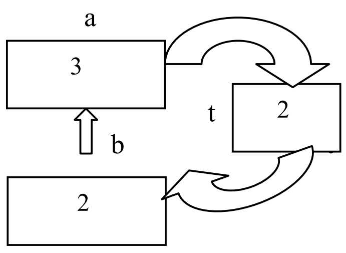
OUTPUT

```
Values before swap are 2   3
Values after swap are 2    3
```

But the output should have been 3   2. So what happened?



Values passing from main ( ) to swap() function



Variables in swap ( ) function

Here we observe that the changes which takes place in argument variables are not reflected in the main() function; as these variables namely a, b and t will be destroyed with function return.

- All these disadvantages will be removed by using *"call by reference"*, which will be discussed with the introduction of pointers in UNIT 11.

Check Your Progress 3

1. Write a function to print Fibonacci series upto 'n' terms 1,1,2,3,…..n

   ……………………………………………………………………………………

   ……………………………………………………………………………………

2. Write a function power (a, b) to calculate a^b

   ……………………………………………………………………………………

   ……………………………………………………………………………………

   ……………………………………………………………………………………

8.9 RECURSION

Within a function body, if the function calls itself, the mechanism is known as **'Recursion'** and the function is known as **'Recursive function'**. Now let us study this mechanism in detail and understand how it works.

- As we see in this mechanism, a chaining of function calls occurs, so it is necessary for a recursive function to stop somewhere or it will result into infinite callings. So the most important thing to remember in this mechanism is that every "recursive function" should have a terminating condition.
- Let us take a very simple example of calculating factorial of a number, which we all know is computed using this formula 5! = 5*4*3*2*1
- First we will write non – recursive or iterative function for this.

Example 8.13

Write a program to find factorial of a number

```
#include <stdio.h>
main ()
{
int n, factorial;
int fact(int);
printf ("Enter any number:\n" );
scanf ("%d", &n);
factorial = fact ( n);   /* function call  */
 printf ("Factorial is %d\n", factorial);
}

/* Non recursive function of factorial */

int fact (int n)
{
int res = 1, i;
for (i = n; i >= 1; i--)
res = res * i;
return (res);
}
```

OUTPUT

```
Enter any number: 5
Factorial is 120
```

How it works?

Suppose we call this function with n = 5

Iterations:

1. i= 5 res = 1*5 = 5
2. i= 4 res = 5*4 = 20
3. i= 3 res = 20*4 = 60
4. i= 2 res = 60*2 = 120
5. i= 1 res = 120*1 = 120

Now let us write this function **recursively**. Before writing any function recursively, we first have to examine the problem, that it can be implemented through recursion.

For instance, we know n! = n* (n – 1)! (Mathematical formula)

Or fact (n) = n*fact (n-1)
Or fact (5) = 5*fact (4)

That means this function calls itself but with value of argument *decreased by '1'*.

Example 8.14

Modify the program 8 using recursion.

```
/*Program to find factorial using recursion*/
#include<stdio.h>
main()
{
int n, factorial;
int fact(int);
printf("Enter any number:  \n" );
scanf("%d",&n);
factorial = fact(n);          /*Function call  */
printf ("Factorial is %d\n", factorial);        }

/* Recursive function of factorial */
int fact(int n)
{
int res;
        if(n == 1)              /* Terminating condition  */
                return(1);
        else
                 res = n*fact(n-1);       /* Recursive call */
                return(res);  }
```

OUTPUT

```
Enter any number: 5
Factorial is 120
```

How it works?

Suppose we will call this function with n = 5

$$\begin{tikzcd}
\text{fact (5)} \arrow[d] & 120 & {} & \\
\text{5*fact(4)} \arrow[d] & \text{5*24} \arrow[u] & & \text{Returning Process} \\
\text{4*fact(3)} \arrow[d] & \text{4*6(=24)} \arrow[u] & & \\
\text{3*fact(2)} \arrow[d] & \text{3*2*1(= 6)} \arrow[u] & & \\
\text{2*fact(1)} \arrow[d] & \text{2* 1(=2)} \arrow[u] & {} \arrow[uuuu] & \\
\text{2*1} & \text{(It terminates here)} & &
\end{tikzcd}$$

Thus a recursive function first proceeds towards the innermost condition, which is the termination condition, and then returns with the value to the outermost call and produces result with the values from the previous return.

Note: This mechanism applies only to those problems, which repeats itself. These types of problems can be implemented either through loops or recursive functions, which one is better understood to you.

Check Your Progress 4

1. Write recursive functions for calculating power of a number 'a' raised by another number 'b' i.e. a^b

   ………………………………………………………………………………………

   ………………………………………………………………………………………

   ………………………………………………………………………………………

   ……………………………………………………………………………..........

8.10 SUMMARY

In this unit, we learnt about "Functions": definition, declaration, prototypes, types, function calls datatypes and storage classes, types function invoking and lastly Recursion. All these subtopics must have given you a clear idea of how to create and call functions from other functions, how to send values through arguments, and how to return values to the called function. We have seen that the functions, which do not return any value, must be declared as *"void"*, return type. A function can return only one value at a time, although it can have many return statements. A function can return any of the data type specified in 'C'.

Any variable declared in functions are local to it and are created with function call and destroyed with function return. The actual and formal arguments should match in type, order and number. A recursive function should have a terminating condition i.e. function should return a value instead of a repetitive function call.

8.11 SOLUTIONS / ANSWERS

Check Your Progress 1

1.
```
/* Function to multiply two integers */
int mul( int a, int b)
{
    int c;
     c = a*b;
     return( c );
}
```

2.
```
#include <stdio.h>
main ( )
{
int x, y, z;
int mul (int, int);     /* function prototype */
printf ("Enter two numbers");
scanf ("%d %d", &x, &y);
z = mul (x, y);                  /* function call  */
printf ("result is %d", z);      }
```

Check Your Progress 2

1. (a) Valid
   (b) In valid
   (c) Valid
   (d) Valid
   (e) Invalid

Check Your Progress 3

1. /* Function to print Fibonacci Series */

```
void fib(int n)
   {
      int curr_term, int count = 0;
      int first = 1;
      int second = 1;
      print ("%d %d", curr_term);
      count = 2;
      while(count < = n)
         {  curr_term = first + second;
            printf ("%d", curr_term);
            first = second;
            second = curr_term;
            count++;
         }
    }
```

2. /* Non Recursive Power function i.e. pow(a, b) */

```
int pow( int a, int b)
     {
 int i, p = 1;
 for (i = 1; i < = b; i++)
 p = p*a;
 return (p);
        }
```

Check Your Progress 4

1. /* Recursive Power Function */

```
 int pow ( int a, int b )
   {       if ( b = = 0 )
               return (1);
          else
               return (a* pow (a, b-1 ));     /*  Recursive call  */
    }

 /*  Main Function  */
 main ( )
        {
              int a, b, p;
              printf (" Enter two numbers");
              scanf ( "%d %d", &a, &b );
              p = pow (a, b);      /* Function call */
printf ( " The result is %d", p);
        }
```

8.12 FURTHER READINGS

1. The C programming language, *Brain W. Kernighan, Dennis M. Ritchie*, PHI
2. C,The Complete Reference, Fourth Edition, *Herbert Schildt*, Tata McGraw Hill, 2002.
3. Computer Programming in C, *Raja Raman. V*, 2002, PHI.
5. C,The Complete Reference, Fourth Edition, *Herbert Schildt*, TMGH,2002.

UNIT 9 STRUCTURES AND UNIONS

Structure

9.0 INTRODUCTION

We have seen so far how to store numbers, characters, strings, and even large sets of these primitives using arrays, but what if we want to store collections of different kinds of data that are somehow related. For example, a file about an employee will probably have his/her name, age, the hours of work, salary, etc. Physically, all of that is usually stored in someone's filing cabinet. In programming, if you have lots of related information, you group it together in an organized fashion. Let's say you have a group of employees, and you want to make a database! It just wouldn't do to have tons of loose variables hanging all over the place. Then we need to have a single data entity where we will be able to store all the related information together. But this can't be achieved by using the arrays alone, as in the case of arrays, we can group multiple data elements that are of the same data type, and is stored in consecutive memory locations, and is individually accessed by a subscript. That is where the user-defined datatype *Structures* come in.

Structure is commonly referred to as a user-defined data type. C's *structures* allow you to store multiple variables of any type in one place (the structure). A structure can contain any of C's data types, including arrays and other structures. Each variable within a structure is called a *member* of the structure. They can hold any number of variables, and you can make arrays of structures. This flexibility makes structures ideally useful for creating databases in C. Similar to the structure there is another user defined data type called *Union* which allows the programmer to view a single storage in more than one way i.e., a variable declared as union can store within its storage space, the data of different types, at different times. In this unit, we will be discussing the user-defined data type structures and unions.

9.1 OBJECTIVES

After going through this unit you should be able to:

- declare and initialize the members of the structures;
- access the members of the structures;
- pass the structures as function arguments;
- declare the array of structures;
- declare and define union; and
- perform all operations on the variables of type Union.

9.2 DECLARATION OF STRUCTURES

To declare a structure you must start with the keyword ***struct*** followed by the *structure name* or *structure tag* and within the braces the list of the structure's member variables. Note that the structure declaration does not actually create any variables. The syntax for the structure declaration is as follows:

```
struct structure-tag {
                    datatype variable1;
                    datatype variable2;
                    dataype variable 3;
                        ...
                    };
```

For example, consider the student database in which each student has a roll number, name and course and the marks obtained. Hence to group this data with a structure-tag as ***student***, we can have the declaration of structure as:

```
struct student {
                int roll_no;
                char name[20];
                char course[20];
                int marks_obtained ;
              };
```

The point you need to remember is that, till this time no memory is allocated to the structure. This is only the definition of structure that tells us that there exists a user-defined data type by the name of student which is composed of the following members. Using this structure type, we have to create the structure variables:

```
 struct student stud1, stud2 ;
```

At this point, we have created two instances or structure variables of the user-defined data type student. Now memory will be allocated. The amount of memory allocated will be the sum of all the data members which form part of the structure template.

The second method is as follows:

```
struct {
        int roll_no;
        char name[20];
        char course[20];
        int marks_obtained ;
     }  stud1, stud2 ;
```

In this case, a tag name *student* is missing, but still it happens to be a valid declaration of structure. In this case the two variables are allocated memory equivalent to the members of the structure.

The advantage of having a tag name is that we can declare any number of variables of the tagged named structure later in the program as per requirement.

If you have a small structure that you just want to define in the program, you can do the definition and declaration together as shown below. This will define a structure of type *struct telephone* and declare three instances of it.

Consider the example for declaring and defining a structure for the telephone billing with three instances:

```
struct telephone{
                    int tele_no;
                    int cust_code;
```

```
                char cust_address[40];
                int bill_amt;
        }       tele1, tele2, tele3;
```

The structure can also be declared by using the typedefinition or typedef. This can be done as shown below:

```
typedef struct country{
                char name[20];
                int population;
                char language[10];
              } Country;
```

This defines a structure which can be referred to either as *struct country* or *Country*, whichever you prefer. Strictly speaking, you don't need a tag name both before and after the braces if you are not going to use one or the other. But it is a standard practice to put them both in and to give them the same name, but the one after the braces starts with an uppercase letter.

The *typedef* statement doesn't occupy storage: it simply defines a new type. Variables that are declared with the *typedef* above will be of type *struct country,* just like population is of type integer. The structure variables can be now defined as below:

Country Mexico, Canada, Brazil;

9.3 ACCESSING THE MEMBERS OF A STRUCTURE

Individual structure members can be used like other variables of the same type. Structure members are accessed using *the structure member operator* (.), also called the *dot operator,* between the structure name and the member name. The syntax for accessing the member of the structure is:

structurevariable. member-name;

Let us take the example of the coordinate structure.

```
struct coordinate{
              int x;
              int y;
        };
```

Thus, to have the structure named first refer to a screen location that has coordinates

x=50, y=100, you could write as,

```
first.x = 50;
first.y = 100;
```

To display the screen locations stored in the structure second, you could write,

```
printf ("%d,%d", second.x, second.y);
```

The individual members of the structure behave like ordinary date elements and can be accessed accordingly.

Now let us see the following program to clarify our concepts. For example, let us see, how will we go about storing and retrieving values of the individual data members of the student structure.

Example 9.1

/*Program to store and retrieve the values from the student structure*/

```
#include<stdio.h>
struct student {
                int roll_no;
                char name[20];
                char course[20];
                int marks_obtained ;
        };
main()
{
student s1 ;
printf (“Enter the  student roll number:”);
scanf (“%d”,&s1.roll_no);
printf (“\nEnter the student name: “);
scanf (“%s”,s1.name);
printf (“\nEnter the student  course”);
scanf (“%s”,s1.course);
printf (“Enter the student percentage\n”);
scanf (“%d”,&s1.marks_obtained);
printf (“\nData entry is complete”);
printf ( “\nThe data entered is as follows:\n”);
printf (“\nThe student roll no is %d”,s1.roll_no);
printf (“\nThe student name is %s”,s1.name);
printf (“\nThe student course is %s”,s1.course);
printf (“\nThe student percentage is %d”,s1.marks_obtained);
}
```

OUTPUT

```
Enter the student roll number: 1234
Enter the student name: ARUN
Enter the student course: MCA
Enter the student percentage: 84
Date entry is complete

The data entered is as follows:
The student roll no is 1234
The student name is ARUN
The student course is MCA
The student percentage is 84
```

Another way of accessing the storing the values in the members of a structure is by initializing them to some values at the time when we create an instance of the data type.

9.4 INITIALIZING STRUCTURES

Like other C variable types, structures can be initialized when they’re declared. This procedure is similar to that for initializing arrays. The structure declaration is followed by an equal sign and a list of initialization values is separated by commas and enclosed in braces. For example, look at the following statements for initializing the values of the members of the *mysale* structure variable.

Example 9.2

```
struct sale {
          char customer[20];
         char item[20];
          float amt;
           } mysale = { "XYZ Industries",
                        “toolskit",
                        600.00
                       };
```

In a structure that contains structures as members, list the initialization values in order. They are placed in the structure members in the order in which the members are listed in the structure definition. Here’s an example that expands on the previous one:

Example 9.3

```
struct customer {
                char firm[20];
                char contact[25];
                }

struct sale {
           struct customer buyer1;
           char item [20];
           float amt;
           } mysale = {
                        { "XYZ Industries", "Tyran Adams"},
                        "toolskit",
                        600.00
                       };
```

These statements perform the following initializations:

- the structure member *mysale.buyer1.firm* is initialized to the string “XYZ Industries”.
- the structure member *mysale.buyer1.contact* is initialized to the string “Tyran Adams”.
- the structure member *mysale.item* is initialized to the string "toolskit".
- the structure member *mysale.amount* is initialized to the amount 600.00.

For example let us consider the following program where the data members are initialized to some value.

Example 9.4

Write a program to access the values of the structure initialized with some initial values.

```
/* Program to illustrate to access the values of the structure initialized with some
initial values*/

#include<stdio.h>
struct telephone{
        int tele_no;
        int cust_code;
        char cust_name[20];
        char cust_address[40];
        int bill_amt;
        };
main()
{
 struct telephone tele = {2314345,
                        5463,
                        "Ram",
                        "New Delhi",
                    2435    };

printf("The values are initialized in this program.");
printf("\nThe telephone number is  %d",tele.tele_no);
printf("\nThe customer code is  %d",tele.cust_code);
printf("\nThe customer name is  %s",tele.cust_name);
printf("\nThe customer address is  %s",tele.cust_address);
printf("\nThe bill amount is   %d",tele.bill_amt);
}
```

OUTPUT

```
The values are initialized in this program.
The telephone number is 2314345
The customer code is 5463
The customer name is Ram
The customer Address is New Delhi
The bill amount is 2435
```

Check Your Progress 1

1. What is the difference between the following two declarations?
    struct x1{……….};
    typedef struct{………}x2;

………………………………………………………………………………………
………………………………………………………………………………………

2. Why can't you compare structures?

………………………………………………………………………………………
………………………………………………………………………………………..

3. Why does size of report a larger size than, one expects, for a structure type, as if there were padding at the end?

………………………………………………………………………………………
………………………………………………………………………………………

4. Declare a structure and instance together to display the date.

………………………………………………………………………………………
………………………………………………………………………………………

9.5 STRUCTURES AS FUNCTION ARGUMENTS

C is a structured programming language and the basic concept in it is the modularity of the programs. This concept is supported by the functions in C language. Let us look into the techniques of passing the structures to the functions. This can be achieved in primarily two ways: Firstly, to pass them as simple parameter values by passing the structure name and secondly, through pointers. We will be concentrating on the first method in this unit and passing using pointers will be taken up in the next unit. Like other data types, a structure can be passed as an argument to a function. The program listing given below shows how to do this. It uses a function to display data on the screen.

Example 9.5

Write a program to demonstrate passing a structure to a function.

```
/*Program to demonstrate passing a structure to a function.*/

#include <stdio.h>

/*Declare and define a structure to hold the data.*/

struct data{
        float amt;
        char fname [30];
        char lname [30];
        } per;

main()
{
void print_per (struct data x);
printf("Enter the donor's first and last names separated by a space:");
scanf ("%s %s", per.fname, per.lname);
printf ("\nEnter the amount donated in rupees:");
scanf ("%f", &per.amt);
print_per (per);
return 0;
 }

void print_per(struct data x)
{
  printf ("\n %s %s gave donation of amount Rs.%.2f.\n", x.fname, x.lname, x.amt);
}
```

OUTPUT

```
Enter the donor's first and last names separated by a space: RAVI KANT
Enter the amount donated in rupees: 1000.00
RAVI KANT gave donation of the amount Rs. 1000.00.
```

You can also pass a structure to a function by passing the structure's address (that is, a pointer to the structure which we will be discussing in the next unit). In fact, in the older versions of C, this was the only way to pass a structure as an argument. It is not necessary now, but you might see the older programs that still use this method. If you pass a pointer to a structure as an argument, remember that you must use the indirect membership operator (→) to access structure members in the function.

Please note the following points with respect to passing the structure as a parameter to a function.

- The return value of the called function must be declared as the value that is being returned from the function. If the function is returning the entire structure then the return value should be declared as *struct* with appropriate tag name.
- The actual and formal parameters for the structure data type must be the same as the *struct* type.
- The return statement is required only when the function is returning some data.
- When the return values of type is *struct*, then it must be assigned to the structure of identical type in the calling function.

Let us consider another example as shown in the Example 9.6, where *structure salary* has three fields related to an employee, namely - *name, no_days_worked* and d*aily_wage*. To accept the values from the user we first call the function *get_data* that gets the values of the members of the structure. Then using the *wages* function we calculate the salary of the person and display it to the user.

Example 9.6

Write a program to accept the data from the user and calculate the salary of the person using concept of functions.

```
/* Program to accept the data from the user and calculate the salary of the person*/

#include<stdio.h>
main()
{
   struct sal      {
            char name[30];
            int no_days_worked;
            int daily_wage;      };
            struct sal salary;
            struct sal get_dat(struct);            /* function prototype*/
            float  wages(struct);                  /*function prototype*/
            float amount_payable;                  /* variable declaration*/
            salary = get_data(salary);
            printf("The name of employee is %s",salary.name);
            printf("Number of days worked is %d",salary.no_daya_worked);
            printf("The daily wage of the employees is %d",salary.daily_wage);
           amount_payable = wages(salary);
            printf("The amount payable to %s is %f",salary.name,amount_payable);
}

struct sal get_data(struct sal income)
            {
            printf("Please enter the employee name:\n");
            scanf("%s",income.name);
            printf("Please enter the number of days worked:\n");
            scanf("%d",&income.no_days_worked);
            printf('Please enter the employee daily wages:\n");
            scanf("%d",&income.daily_wages);
           return(income);
          }

float wages(struct)
{
   struct sal amt;
   int total_salary ;
   total_salary = amt.no_days_worked * amt.daily_wages;
   return(total_salary);     }
```

Check Your Progress 2

1. How is structure passing and returning implemented?

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

2. How can I pass constant values to functions which accept structure arguments?

…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………

3. What will be the output of the program?

```
#include<stdio.h>
main( )
{
struct pqr{
      int x ;
     };
struct pqr  pqr ;
pqr.x =10 ;
printf ("%d", pqr.x);
}
```

…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………

9.6 STRUCTURES AND ARRAYS

Thus far we have studied as to how the data of heterogeneous nature can be grouped together and be referenced as a single unit of structure. Now we come to the next step in our real world problem. Let's consider the example of students and their marks. In this case, to avoid declaring various data variables, we grouped together all the data concerning the student's marks as one unit and call it student. The problem that arises now is that the data related to students is not going to be of a single student only. We will be required to store data for a number of students. To solve this situation one way is to declare a structure and then create sufficient number of variables of that structure type. But it gets very cumbersome to manage such a large number of data variables, so a better option is to declare an array.

So, revising the array for a few moments we would refresh the fact that an array is simply a collection of homogeneous data types. Hence, if we make a declaration as:

int temp[20];

It simply means that temp is an array of twenty elements where each element is of type integer, indicating homogenous data type. Now in the same manner, to extend the concept a bit further to the structure variables, we would say,

struct student   stud[20] ;

It means that *stud* is an array of twenty elements where each element is of the type *struct student* (which is a user-defined data type we had defined earlier). The various members of the *stud* array can be accessed in the similar manner as that of any other ordinary array.

For example,
struct student stud[20], we can access the *roll_no* of this array as

stud[0].roll_no;
stud[1].roll_no;
stud[2].roll_no;
stud[3].roll_no;

```
…
…
…
stud[19].roll_no;
```

Please remember the fact that for an array of twenty elements the subscripts of the array will be ranging from 0 to 19 (a total of twenty elements). So let us now start by seeing how we will write a simple program using array of structures.

Example 9.7

Write a program to read and display data for 20 students.

```
/*Program to read and print the data for 20 students*/

#include <stdio.h>
struct student {  int roll_no;
                  char name[20];
                  char course[20];
                  int marks_obtained ;
                 };
main( )
{
 struct student stud [20];
 int i;
 printf (“Enter the student data one by one\n”);
 for(i=0; i<=19; i++)
  {
     printf (“Enter the roll number of %d student”,i+1);
     scanf (“%d”,&stud[i].roll_no);
     printf (“Enter the name of %d student”,i+1);
     scanf (“%s”,stud[i].name);
     printf (“Enter the course of %d student”,i+1);
     scanf (“%d”,stud[i].course);
     printf (“Enter the marks obtained of %d student”,i+1);
     scanf (“%d”,&stud[i].marks_obtained);
   }
     printf (“the data entered is as follows\n”);
     for (i=0;i<=19;i++)
  {
     printf (“The roll number of  %d student is %d\n”,i+1,stud[i].roll_no);
     printf (“The name of  %d student is %s\n”,i+1,stud[i].name);
     printf (“The course of  %d student is %s\n”,i+1,stud[i].course);
     printf (“The marks of  %d student is %d\n”,i+1,stud[i].marks_obtained);
    }
}
```

The above program explains to us clearly that the array of structure behaves as any other normal array of any data type. Just by making use of the subscript we can access all the elements of the structure individually.

Extending the above concept where we can have arrays as the members of the structure. For example, let’s see the above example where we have taken a structure for the student record. Hence in this case it is a real world requirement that each student will be having marks of more than one subject. Hence one way to declare the structure, if we consider that each student has 3 subjects, will be as follows:

```
struct student {
                  int roll_no;
                  char name [20];
```

```
                 char course [20];
                 int subject1 ;
                 int subject2;
                 int subject3;
                };
```

The above described method is rather a bit cumbersome, so to make it more efficient we can have an array inside the structure, that is, we have an array as the member of the structure.

```
struct student {
                 int roll_no;
                 char name [20];
                 char course [20];
                 int subject [3] ;
                };
```

Hence to access the various elements of this array we can the program logic as follows:

Example 9.8

```
/*Program to read and print data related to five students having marks of three
subjects each using the concept of arrays */

#include<stdio.h>
struct student {
                 int roll_no;
                 char name [20];
                 char course [20];
                 int subject [3] ;
                };
main( )
{
  struct student stud[5];
  int i,j;
printf (“Enter the data for all the students:\n”);
for (i=0;i<=4;i++)
{
printf (“Enter the roll number of %d student”,i+1);
scanf (“%d”,&stud[i].roll_no);
printf(“Enter the name of %d student”,i+1);
scanf (“%s”,stud[i].name);
printf (“Enter the course of %d student”,i+1);
scanf (“%s”,stud[i].course);
for (j=0;j<=2;j++)
   {
    printf (“Enter the marks of the %d subject of the student %d:\n”,j+1,i+1);
   scanf (“%d”,&stud[i].subject[j]);
   }
}
printf (“The data you have entered is as follows:\n”);
for (i=0;i<=4;i++)
    {
        printf (“The %d th student's roll number is %d\n”,i+1,stud[i].roll_no);
        printf (“The %d the student's name is %s\n”,i+1,stud[i].name);
        printf (“The %d the student's course is %s\n”,i+1,stud[i].course);
        for (j=0;j<=2;j++)
        {
        printf (“The %d the student's marks of %d   I  subject are %d\n”,i+1, j+1,
        stud[i].subject[j]);
```

```
            }
    }
   printf ("End of the program\n");
}
```

Hence as described in the example above, the array as well as the arrays of structures can be used with efficiency to resolve the major hurdles faced in the real world programming environment.

9.7 UNIONS

Structures are a way of grouping homogeneous data together. But it often happens that at any time we require only one of the member's data. For example, in case of the support price of shares you require only the latest quotations. And only the ones that have changed need to be stored. So if we declare a structure for all the scripts, it will only lead to crowding of the memory space. Hence it is beneficial if we allocate space to only one of the members. This is achieved with the concepts of the *UNIONS*. *UNIONS* are similar to *STRUCTURES* in all respects but differ in the concept of storage space.

A *UNION* is declared and used in the same way as the structures. Yet another difference is that only one of its members can be used at any given time. Since all members of a Union occupy the same memory and storage space, the space allocated is equal to the largest data member of the Union. Hence, the member which has been updated last is available at any given time.

For example a union can be declared using the syntax shown below:

```
union union-tag {
                        datatype variable1;
                        datatype variable2;
                            ...
                    };
```

For example,

```
union temp{
             int   x;
             char y;
             float z;
            };
```

In this case a float is the member which requires the largest space to store its value hence the space required for float (4 bytes) is allocated to the union. All members share the same space. Let us see how to access the members of the union.

Example 9.9

Write a program to illustrate the concept of union.

```
/* Declare a union template called tag */
union tag {
            int nbr;
            char character;
            }
/* Use the union template */
union tag mixed_variable;
/* Declare a union and instance together */
union generic_type_tag {
                            char c;
                            int i;
float f;
double d;
} generic;
```

9.8 INITIALIZING AN UNION

Let us see, how to initialize a Union with the help of the following example:

Example 9.10

```
union date_tag {
            char complete_date [9];
            struct part_date_tag {
                        char month[2];
                        char break_value1;
                        char day[2];
                        char break_value2;
                        char year[2];
                            } parrt_date;
            }date = {"01/01/05"};
```

9.9 ACCESSING THE MEMBERS OF AN UNION

Individual union members can be used in the same way as the structure members, by using the member operator or dot operator (.). However, there is an important difference in accessing the union members. Only one union member should be accessed at a time. Because a union stores its members on top of each other, it's important to access only one member at a time. Trying to access the previously stored values will result in erroneous output.

Check Your Progress 3

1. What will be the output?

```
  #include<stdio.h>
  main()
  {
  union{
        struct{
             char x;
             char y;
             char z;
             char w;
            }xyz;

        struct{
            int p;
            int q ;
            }pq;
            long  a ;
            float b;
            double d;
            }prq;
        printf ("%d",sizeof(prq));
             }
```

9.10 SUMMARY

In this unit, we have learnt how to use structures, a data type that you design to meet the needs of a program. A structure can contain any of C's data types, including other structures, pointers, and arrays. Each data item within a structure, called a *member,* is accessed using the structure member operator (.) between the structure name and the member name. Structures can be used individually, and can also be used in arrays.

Unions were presented as being similar to structures. The main difference between a union and a structure is that the union stores all its members in the same area. This means that only one member of a union can be used at a time.

9.11 SOLUTIONS / ANSWERS

Check Your Progress 1

1. The first form declares a *structure tag*; the second declares a *typedef*. The main difference is that the second declaration is of a slightly more abstract type - users do not necessarily know that it is a structure, and the keyword struct is not used while declaring an instance.

2. There is no single correct way for a compiler to implement a structure comparison consistent with C's low-level flavor. A simple byte-by-byte comparison could detect the random bits present in the unused "holes" in the structure (such padding is used to keep the alignment of later fields correct). A field-by-field comparison for a large structure might require an inordinate repetitive code.

3. Structures may have this padding (as well as internal padding), to ensure that alignment properties will be preserved when an array of contiguous structures is allocated. Even when the structure is not part of an array, the end padding remains, so that *sizeof* can always return a consistent size.

4.
```
struct date {
            char month[2];
            char day[2];
            char year[4];
          } current_date;
```

Check Your Progress 2

1. When structures are passed as arguments to functions, the entire structure is typically pushed on the stack, using as many words. (Programmers often choose to use pointers instead, to avoid this overhead). Some compilers merely pass a pointer to the structure, though they may have to make a local copy to preserve pass-by value semantics.

   Structures are often returned from functions in a pointed location by an extra, compiler-supplied "hidden" argument to the function. Some older compilers used a special, static location for structure returns, although this made structure - valued functions non-reentrant, which ANSI C disallows.

2. C has no way of generating anonymous structure values. You will have to use a temporary structure variable or a little structure - building function.

3. 10

Check Your Progress 3

1. 8

9.12 FURTHER READINGS

1. The C Programming Language, *Kernighan & Richie*, PHI Publication, 2002.
2. Computer Science A structured programming approach using C, *Behrouza .Forouzan, Richard F. Gilberg*, Second Edition, Brooks/Cole, Thomson Learning, 2001.
3. Programming with C, Schaum Outlines, Second Edition, *Gottfried*, Tata McGraw Hill, 2003.

UNIT 10 POINTERS

Structure

10.0 INTRODUCTION

If you want to be proficient in the writing of code in the C programming language, you must have a thorough working knowledge of how to use pointers. One of those things, beginners in C find difficult is the concept of pointers. The purpose of this unit is to provide an introduction to pointers and their efficient use in the C programming. Actually, the main difficulty lies with the C's pointer terminology than the actual concept.

C uses pointers in three main ways. First, they are used to create *dynamic data structures*: data structures built up from blocks of memory allocated from the heap at run-time. Second, C uses pointers to handle *variable parameters* passed to functions. And third, pointers in C provide an alternative means of accessing information stored in arrays, which is especially valuable when you work with strings.

A normal variable is a location in memory that can hold a value. For example, when you declare a variable *i* as an integer, four bytes of memory is set aside for it. In your program, you refer to that location in memory by the name *i*. At the machine level, that location has a memory address, at which the four bytes can hold one integer value. A *pointer* is a variable that points to another variable. This means that it holds the memory address of another variable. Put another way, the pointer does not hold a value in the traditional sense; instead, it holds the address of another variable. It points to that other variable by holding its address.

Because a pointer holds an address rather than a value, it has two parts. The pointer itself holds the address. That addresses points to a value. There is the pointer and the value pointed to. As long as you're careful to ensure that the pointers in your programs always point to valid memory locations, pointers can be useful, powerful, and relatively trouble-free tools.

We will start this unit with a basic introduction to pointers and the concepts surrounding pointers, and then move on to the three techniques described above. Thorough knowledge of the *pointers* is very much essential for your future courses like the *datastructures, design and analysis of algorithms etc..*

10.1 OBJECTIVES

After going through this unit you should be able to:

- understand the concept and use pointers;
- address and use of indirection operators;
- make pointer type declaration, assignment and initialization;
- use null pointer assignment;
- use the pointer arithmetic;
- handle pointers to functions;
- see the underlying unit of arrays and pointers; and
- understand the concept of dynamic memory allocation.

10.2 POINTERS AND THEIR CHARACTERISTICS

Computer's memory is made up of a sequential collection of storage cells called bytes. Each byte has a number called an address associated with it. When we declare a variable in our program, the compiler immediately assigns a specific block of memory to hold the value of that variable. Since every cell has a unique address, this block of memory will have a unique starting address. The size of this block depends on the range over which the variable is allowed to vary. For example, on 32 bit PC's the size of an integer variable is 4 bytes. On older 16 bit PC's integers were 2 bytes. In C the size of a variable type such as an integer need not be the same on all types of machines. If you want to know the size of the various data types on your system, running the following code given in the Example 10.1 will give you the information.

Example 10.1

Write a program to know the size of the various data types on your system.

```
# include <stdio.h>
main( )
{
  printf (“n Size of a int = %d bytes”, sizeof (int));
  printf (“\n Size of a float = %d bytes”, sizeof (float));
  printf (“\n Size of a char = %d bytes”, sizeof (char));
}
```

OUTPUT

```
Size of int = 2 bytes
Size of float = 4 bytes
Size of char = 1 byte
```

An *ordinary variable* is a location in memory that can hold a value. For example, when you declare a variable *num* as an integer, the compiler sets aside 2 bytes of memory (depends up the PC) to hold the value of the integer. In your program, you refer to that location in memory by the name *num*. At the machine level that location has a memory address.

int num = 100;

We can access the value 100 either by the name num or by its memory address. Since addresses are simply digits, they can be stored in any other variable. Such variables that hold addresses of other variables are called *Pointers*. In other words, a *pointer* is simply a variable that contains an address, which is a location of another variable in memory. A pointer variable "points to" another variable by holding its address. Since a pointer holds an address rather than a value, it has two parts. The pointer itself holds the address. That addresses points to a value. There is a pointer and the value pointed to. This fact can be a little confusing until you get comfortable with it, but once you get familiar with it, then it is extremely easy and very powerful. One good way to visualize this concept is to examine the figure 10.1 given below:

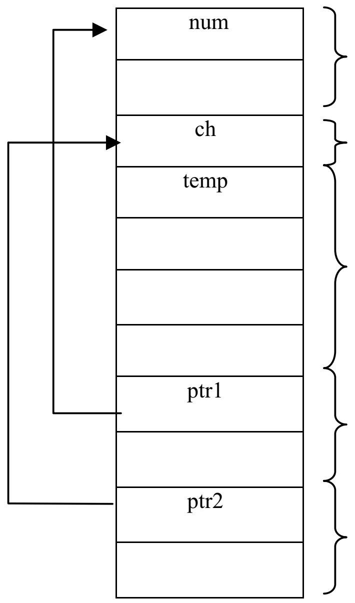


Figure 10.1: Concept of pointer variables

Let us see the important features of the pointers as follows:

Characteristic features of Pointers:

With the use of pointers in programming,

i. The program execution time will be faster as the data is manipulated with the help of addresses directly.
ii. Will save the memory space.
iii. The memory access will be very efficient.
iv. Dynamic memory is allocated.

10.3 THE ADDRESS AND INDIRECTION OPERATORS

Now we will consider how to determine the address of a variable. The operator that is available in C for this purpose is "&" (*address of*) operator. The operator & and the immediately preceding variable returns the address of the variable associated with it. C's other unary pointer operator is the "*", also called as *value at address* or indirection operator. It returns a value stored at that address. Let us look into the illustrative example given below to understand how they are useful.

Example 10.2

Write a program to print the address associated with a variable and value stored at that address.

/* Program to print the address associated with a variable and value stored at that address*/

```
# include <stdio.h>
main( )
{
  int qty = 5;
  printf ("Address of qty = %u\n",&qty);
  printf ("Value of qty = %d \n",qty);
  printf("Value of qty = %d",*(&qty));
 }
```

OUTPUT

```
Address of qty = 65524
Value of qty = 5
Value of qty = 5
```

Look at the *printf* statement carefully. The format specifier *%u* is taken to increase the range of values the address can possibly cover. The system-generated address of the variable is not fixed, as this can be different the next time you execute the same program. Remember unary operator operates on single operands. When *&* is preceded by the variable qty, has returned its address. Note that the *&* operator can be used only with simple variables or array elements. It cannot be applied to expressions, constants, or register variables.

Observe the third line of the above program. **(&qty)* returns the value stored at address 65524 i.e. 5 in this case. Therefore, *qty* and **(&qty)* will both evaluate to 5.

10.4 POINTER TYPE DECLARATION AND ASSIGNMENT

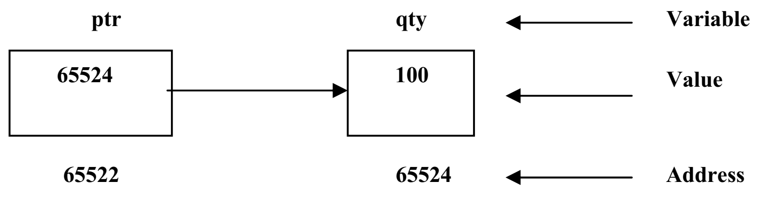
We have seen in the previous section that *&qty* returns the address of *qty* and this address can be stored in a variable as shown below:

ptr = &qty;

In C, every variable must be declared for its data type before it is used. Even this holds good for the pointers too. We know that *ptr* is not an ordinary variable like any integer variable. We declare the data type of the pointer variable as that of the type of the data that will be stored at the address to which it is pointing to. Since *ptr* is a variable, which contains the address of an integer variable *qty*, it can be declared as:

*int *ptr;*

where *ptr* is called a *pointer variable*. In C, we define a pointer variable by preceding its name with an asterisk(*). The “*” informs the compiler that we want a pointer variable, i.e. to set aside the bytes that are required to store the address in memory. The *int* says that we intend to use our pointer variable to store the address of an integer. Consider the following memory map:



Let us look into an example given below:

Example 10.3

```
/* Program below demonstrates the relationships we have discussed so far */

# include <stdio.h>
main( )
{
  int qty = 5;
  int *ptr;         /* declares ptr as a pointer variable that points to an integer variable
*/
  ptr = &qty;   /* assigning qty's address to ptr -> Pointer Assignment */

  printf ("Address of qty = %u \n", &qty);
  printf ("Address of qty = %u \n", ptr);
  printf ("Address of ptr = %u \n", &ptr);
  printf ("Value of ptr = %d \n", ptr);
  printf ("Value of qty = %d \n", qty);
  printf ("Value of qty = %d \n", *(&qty));
  printf ("Value of qty = %d", *ptr);
}
```

OUTPUT

```
Address of qty = 65524
Address of ptr = 65522
Value of ptr = 65524
Value of qty = 5
Value of qty = 5
Value of qty = 5
```

Try this as well:

Example 10.4

```
/* Program that tries to reference the value of a pointer even though the pointer is
uninitialized */

# include <stdio.h>
main()
{
   int *p;   /* a pointer to an integer */
    *p = 10;
   printf("the value is %d", *p);
   printf("the value is %u",p);
}
```

This gives you an error. The pointer *p* is uninitialized and points to a random location in memory when you declare it. It could be pointing into the system stack, or the global variables, or into the program's code space, or into the operating system. When you say **p=10;* the program will simply try to write a 10 to whatever random location ***p*** points to. The program may explode immediately. It may subtly corrupt data in another part of your program and you may never realize it. Almost always, an uninitialized pointer or a bad pointer address causes the fault.

This can make it difficult to track down the error. Make sure you initialize all pointers to a valid address before dereferencing them.

Within a variable declaration, a pointer variable can be initialized by assigning it the address of another variable. Remember the variable whose address is assigned to the pointer variable must be declared earlier in the program. In the example given below, let us assign the pointer *p* with an address and also a value 10 through the **p*.

Example 10.5

Let us say,

```
int x; /* x is initialized to a value 10*/
p = &x;   /* Pointer declaration & Assignment */
*p=10;
```

Let us write the complete program as shown below:

```
# include <stdio.h>
main( )
{
   int *p;   /* a pointer to an integer */
   int x;
   p = &x;
   *p=10;
   printf("The value of x is %d",*p);
   printf("\nThe address in which the x is stored is %d",p);
}
```

OUTPUT

```
The value of x is 10
The address in which the x is stored is 52004
```

This statement puts the value of 20 at the memory location whose address is the value of *px*. As we know that the value of *px* is the address of *x* and so the old value of *x* is replaced by 20. This is equivalent to assigning 20 to *x*. Thus we can change the value of a variable *indirectly* using a pointer and the *indirection operator.*

10.4.1 Pointer to a Pointer

The concept of pointer can be extended further. As we have seen earlier, a pointer variable can be assigned the address of an ordinary variable. Now, this variable itself could be another pointer. This means that a pointer can contain address of another pointer. The following program will makes you the concept clear.

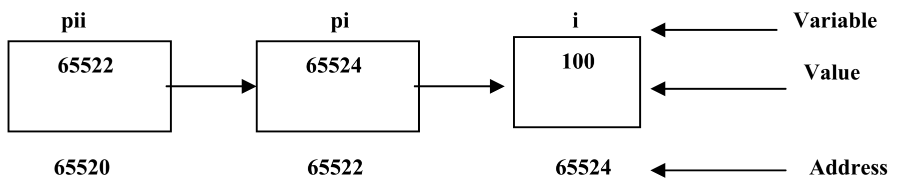
Example 10.6

```
/* Program that declares a pointer to a pointer */

# include<stdio.h>
main( )
{
  int i = 100;
  int *pi;
  int **pii;
  pi = &i;
  pii = &pi;

 printf ("Address of i = %u \n", &i);
 printf ("Address of i = %u \n", pi);
 printf ("Address of i = %u \n", *pii);
 printf ("Address of pi = %u \n", &pi);
 printf ("Address of pi = %u \n", pii);
 printf ("Address of pii = %u \n", &pii);
 printf ("Value of i = %d \n", i);
 printf ("Value of i = %d \n", *(&i));
 printf ("Value of i = %d \n", *pi);
 printf ("Value of i = %d", **pii);
}
```

OUTPUT

```
Address of i = 65524
Address of i = 65524
Address of i = 65524
Address of pi = 65522
Address of pi = 65522
Address of pii = 65520

Value of i = 100
Value of i = 100
Value of i = 100
Value of i = 100
```

Consider the following memory map for the above shown example:



10.4.2 Null Pointer Assignment

It does make sense to assign an integer value to a pointer variable. An exception is an assignment of 0, which is sometimes used to indicate some special condition. A macro is used to represent a null pointer. That macro goes under the name *NULL*. Thus, setting the value of a pointer using the *NULL*, as with an assignment statement such as *ptr = NULL*, tells that the pointer has become a *null* pointer. Similarly, as one can test the condition for an integer value as zero or not, like *if (i == 0)*, as well we can test the condition for a null pointer using *if (ptr == NULL)* or you can even set a pointer to *NULL* to indicate that it's no longer in use. Let us see an example given below.

Example 10.7

```
# include<stdio.h>
# define NULL 0
main()
{
  int *pi = NULL;
  printf("The value of pi is %u", pi);
}
```

OUTPUT

```
The value of pi is 0
```

Check Your Progress 1

1. How is a pointer variable being declared? What is the purpose of data type included in the pointer declaration?

…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………

2. What would be the output of following programs?

```
(i)  void main( )
       {
          int i = 5;
          printf ("Value of i = %d   Address of i = %u", i, &i);
          &i = 65534;
          printf ("\n New value of i = %d  New Address of i = %u", i, &i);
          }

(ii) void main( )
       {
          int *i, *j;
          j = i * 2;
          printf ("j = %u", j);
      }
```

…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………

3. Explain the effect of the following statements:

```
(i)      int x = 10, *px = &x;

(ii)     char *pc;

(iii)    int x;
         void *ptr = &x;
         *(int *) ptr = 10;
```

…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………

10.5 POINTER ARITHMETIC

Pointer variables can also be used in arithmetic expressions. The following operations can be carried out on pointers:

1. Pointers can be incremented or decremented to point to different locations like

```
ptr1 = ptr2 + 3;
ptr ++;
-- ptr;
```

However, *ptr++* will cause the pointer *ptr* to point the next address value of its type. For example, if *ptr* is a pointer to float with an initial value of 65526, then after the operation *ptr* **++** or *ptr* **=** *ptr+1*, the value of *ptr* would be 65530. Therefore, if we increment or decrement a pointer, its value is increased or decreased by the length of the data type that it points to.

2. If *ptr1* and *ptr2* are properly declared and initialized pointers, the following operations are valid:

```
res = res + *ptr1;
*ptr1 = *ptr2 + 5;
prod = *ptr1 * *ptr2;
quo = *ptr1 /  *ptr2;
```

Note that there is a blank space between / and * in the last statement because if you write **/*** together, then it will be considered as the beginning of a comment and the statement will fail.

3. Expressions like *ptr1 == ptr2*, *ptr1 < ptr2*, and *ptr2 != ptr1* are permissible provided the pointers *ptr1* and *ptr2* refer to same and related variables. These comparisons are common in handling arrays.

Suppose *p1* and *p2* are pointers to related variables. The following operations cannot work with respect to pointers:

1. Pointer variables cannot be added. For example, *p1 = p1 + p2* is not valid.

2. Multiplication or division of a pointer with a constant is not allowed. For example, *p1 * p2* or *p2 / 5* are invalid.

3. An invalid pointer reference occurs when a pointer's value is referenced even though the pointer doesn't point to a valid block. Suppose *p* and *q* are two pointers. If we say*, p = q;* when *q* is uninitialized. The pointer *p* will then become uninitialized as well, and any reference to **p* is an invalid pointer reference.

10.6 PASSING POINTERS TO FUNCTIONS

As we have studied in the FUNCITONS that arguments can generally be passed to functions in one of the two following ways:

1. Pass by value method
2. Pass by reference method

In the first method, when arguments are passed by value, a copy of the *values* of actual arguments is passed to the calling function. Thus, any changes made to the variables inside the function will have no effect on variables used in the actual argument list.

However, when arguments are passed by reference (i.e. when a pointer is passed as an argument to a function), the *address* of a variable is passed. The contents of that address can be accessed freely, either in the called or calling function. Therefore, the function called by reference can change the value of the variable used in the call.

Here is a simple program that illustrates the difference.

Example 10.8

Write a program to swap the values using the pass by value and pass by reference methods.

```
 /* Program that illustrates the difference between ordinary arguments, which are
passed by value, and pointer arguments, which are passed by reference */

# include <stdio.h>
main()
{
        int x = 10;
        int y = 20;
       void swapVal ( int, int );           /* function prototype */
       void swapRef ( int *, int * );      /*function prototype*/
        printf("PASS BY VALUE METHOD\n");
        printf ("Before calling function swapVal    x=%d   y=%d",x,y);
        swapVal (x, y);             /* copy of the arguments are passed */
        printf ("\nAfter calling function swapVal   x=%d   y=%d",x,y);
                printf("\n\nPASS BY REFERENCE METHOD");
        printf ("\nBefore calling function swapRef   x=%d   y=%d",x,y);
        swapRef (&x,&y);      /*address of arguments are passed */
        printf("\nAfter calling function swapRef     x=%d   y=%d",x,y);
}

/* Function using the pass by value method*/
void swapVal (int x, int y)
{
        int temp;
        temp = x;
        x = y;
        y = temp;
        printf ("\nWithin function swapVal           x=%d   y=%d",x,y);
        return;
}

/*Function using the pass by reference method*/
void swapRef (int *px, int *py)
{
         int temp;
         temp = *px;
         *px = *py;
         *py = temp;
          printf ("\nWithin function swapRef       *px=%d   *py=%d",*px,*py);
          return;
}
```

OUTPUT

```
PASS BY VALUE METHOD
Before calling function swapVal   x=10       y=20
Within function swapVal             x=20       y=10
After calling function swapVal      x=10       y=20
```

```
PASS BY REFERENCE METHOD
Before calling function swapRef   x=10      y=20
Within function swapRef          *px=20    *py=10
After calling function swapRef    x=20      y=10
```

This program contains two functions, *swapVal* and *swapRef*.

In the function *swapVal*, arguments *x* and *y* are passed by *value*. So, any changes to the arguments are local to the function in which the changes occur. Note the values of *x* and *y* remain unchanged even after exchanging the values of *x* and *y* inside the function ***swapVal***.

Now consider the function *swapRef*. This function receives two *pointers* to integer variables as arguments identified as pointers by the indirection operators that appear in argument declaration. This means that in the function *swapRef*, arguments *x* and *y* are passed by *reference.* So, any changes made to the arguments inside the function *swapRef* are reflected in the function *main( ).* Note the values of *x* and *y* is interchanged after the function call *swapRef*.

10.6.1 A Function returning more than one value

Using *call by reference* method we can make a function return more than one value at a time, which is not possible in the *call by value* method. The following program will makes you the concept very clear.

Example 10.9

Write a program to find the perimeter and area of a rectangle, if length and breadth are given by the user.

```
/* Program to find the perimeter and area of a rectangle*/

#include <stdio.h>
void main()
{
float len,br;
float peri, ar;
void periarea(float length, float breadth, float *, float *);
printf("\nEnter the length and breadth of  a rectangle in metres: \n");
scanf("%f %f",&len,&br);
periarea(len,br,&peri,&ar);
printf("\nPerimeter of the rectangle is %f metres", peri);
printf("\nArea of the rectangle is %f sq. metres", ar);
}

void periarea(float length, float breadth, float *perimeter, float *area)
{
*perimeter = 2 * (length +breadth);
*area = length * breadth;
}
```

OUTPUT

```
Enter the length and breadth of  a rectangle in metres:
23.0 3.0
Perimeter of the rectangle is 52.000000 metres
Area of the rectangle is 69.000000 sq. metres
```

Here in the above program, we have seen that the function *periarea* is returning two values. We are passing the values of *len* and *br* but, addresses of peri and *ar*. As we are passing the addresses of *peri* and *ar*, any change that we make in values stored at addresses contained in the variables **perimeter* and **area*, would make the change effective even in *main()* also.

10.6.2 Function returning a pointer

A function can also return a pointer to the calling program, the way it returns an int, a float or any other data type. To return a pointer, a function must explicitly mention in the calling program as well as in the function prototype. Let's illustrate this with an example:

Example: 10.10

Write a program to illustrate a function returning a pointer.

```
/*Program that shows how a function returns a pointer */

# include<stdio.h>

void main( )
{
  float *a;
  float *func( );        /* function prototype */
  a = func( );
  printf ("Address = %u", a);
}
float *func( )
{
   float r = 5.2;
   return (&r);
}
```

OUTPUT

```
Address = 65516
```

This program only shows how a function can return a pointer. This concept will be used later while handling arrays.

Check Your Progress 2

1. Tick mark (√)whether each of the following statements are true or false.

(i)	An integer is subtracted from a pointer variable.	☐ True	☐ False
(ii)	Pointer variables can be compared.	☐ True	☐ False
(iii)	Pointer arguments are passed by value.	☐ True	☐ False
(iv)	Value of a local variable in a function can be changed by another function.	☐ True	☐ False
(v)	A function can return more than one value.	☐ True	☐ False
(vi)	A function can return a pointer.	☐ True	☐ False

10.7 ARRAYS AND POINTERS

Pointers and arrays are so closely related. An array declaration such as *int arr[5]* will lead the compiler to pick an address to store a sequence of 5 integers, and *arr* is a name for that address. The array name in this case is the *address* where the sequence of integers starts. Note that the value is not the first integer in the sequence, nor is it the sequence in its entirety. The value is just an address.

Now, if *arr* is a one-dimensional array, then the address of the first array element can be written as *&arr[0]* or simply *arr*. Moreover, the address of the second array element can be written as *&arr[1]* or simply *(arr+1)*. In general, address of array element *(i+1)* can be expressed as either *&arr[i]* or as *(arr+ i)*. Thus, we have two different ways for writing the address of an array element. In the latter case i.e, expression *(arr+ i)* is a symbolic representation for an address rather than an arithmetic expression. Since *&arr[i]* and *(ar+ i)* both represent the address of the *i*th element of *arr,* so *arr[i]* and **(ar + i)* both represent the contents of that address i.e., the value of *i*th element of *arr*.

Note that it is not possible to assign an arbitrary address to an array name or to an array element. Thus, expressions such as *arr*, *(arr+ i)* and *arr[i]* cannot appear on the left side of an assignment statement. Thus we cannot write a statement such as:

```
&arr[0] = &arr[1];      /*  Invalid  */
```

However, we can assign the value of one array element to another through a pointer, for example,

```
ptr = &arr[0];        /*   ptr is a pointer to arr[ 0]  */
arr[1] = *ptr;      /* Assigning the value stored at address to arr[1]  */
```

Here is a simple program that will illustrate the above-explained concepts:

Example 10.11

```
/* Program that accesses array elements of a one-dimensional array using pointers */

# include<stdio.h>
main()
{

  int arr[ 5 ] = {10, 20, 30, 40, 50};
  int i;

  for (i = 0;  i < 5;  i++)
     {
         printf ("i=%d\t arr[i]=%d\t *(arr+i)=%d\t", i, arr[i], *(arr+i));
         printf ("&arr[i]=%u\t arr+i=%u\n", &arr[i], (arr+i));     }
}
```

OUTPUT:

```
i=0     arr[i]=10      *(arr+i)=10   &arr[i]=65516   arr+i=65516
i=1     arr[i]=20      *(arr+i)=20   &arr[i]=65518   arr+i=65518
i=2     arr[i]=30      *(arr+i)=30   &arr[i]=65520   arr+i=65520
i=3     arr[i]=40      *(arr+i)=40   &arr[i]=65522   arr+i=65522
i=4     arr[i]=50      *(arr+i)=50   &arr[i]=65524   arr+i=65524
```

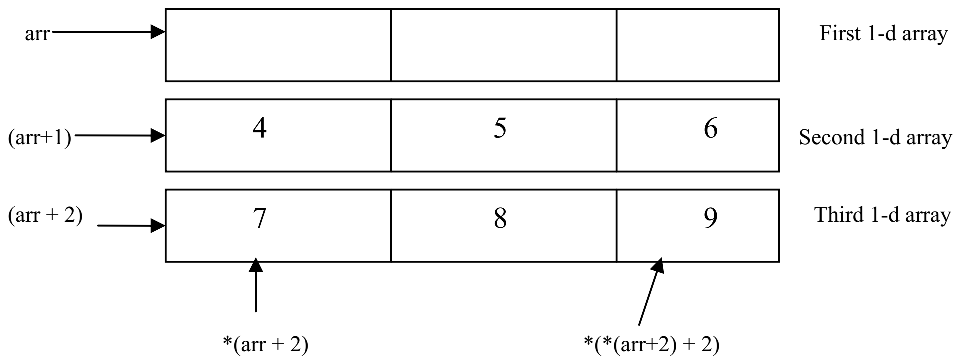
Note that *i* is added to a pointer value (address) pointing to integer data type (i.e., the array name) the result is the pointer is increased by i times the size (in bytes) of integer data type. Observe the addresses 65516, 65518 and so on. So if *ptr* is a char pointer, containing addresses *a*, then *ptr+1* is *a+1*. If *ptr* is a float pointer, then *ptr+ 1* is *a+ 4*.

Pointers and Multidimensional Arrays

C allows multidimensional arrays, lays them out in memory as contiguous locations, and does more behind the scenes address arithmetic. Consider a 2-dimensional array.

int arr[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

The compiler treats a 2 dimensional array as an array of arrays. As you know, an array name is a pointer to the first element within the array. So, **arr** points to the first 3-element array, which is actually the first row (i.e., row 0) of the two-dimensional array. Similarly, *(arr + 1)* points to the second 3-element array (i.e., row 1) and so on. The value of this pointer, *(arr + 1), refers to the entire row. Since row 1 is a one-dimensional array, *(arr + 1)* is actually a pointer to the first element in row 1. Now add 2 to this pointer. Hence, *(*(arr + 1) + 2)* is a pointer to element 2 (i.e., the third element) in row 1. The value of this pointer, **(*(arr + 1) + 2),* refers to the element in column 2 of row 1. These relationships are illustrated below:



10.8 ARRAY OF POINTERS

The way there can be an array of integers, or an array of float numbers, similarly, there can be array of pointers too. Since a pointer contains an address, an array of pointers would be a collection of addresses. For example, a multidimensional array can be expressed in terms of an array of pointers rather than a pointer to a group of contiguous arrays.

Two-dimensional array can be defined as a one-dimensional array of integer pointers by writing:

*int *arr[3];*

rather than the conventional array definition,

int arr[3][5];

Similarly, an n-dimensional array can be defined as (n-1)-dimensional array of pointers by writing

*data-type *arr[subscript 1] [subscript 2]…. [subscript n-1];*

The subscript1, subscript2 indicate the maximum number of elements associated with each subscript.

Example 10.12

Write a program in which a two-dimensional array is represented as an array of integer pointers to a set of single-dimensional integer arrays.

```
/* Program calculates the difference of the corresponding elements of two table of
integers */

# include <stdio.h>
# include <stdlib.h>
# define MAXROWS 3
void main( )
{
    int *ptr1[MAXROWS], *ptr2 [MAXROWS], *ptr3 [MAXROWS];
    int rows, cols, i, j;
    void inputmat (int *[ ], int, int);
    void dispmat (int *[ ], int, int);
    void calcdiff (int *[ ], int *[ ], int *[ ], int, int);

    printf ("Enter no. of rows & columns \n");
    scanf ("%d%d", &rows, &cols);

    for (i = 0; i < rows; i++)
    {
        ptr1[ i ] = (int *) malloc (cols * sizeof (int));
        ptr2[ i ] = (int *) malloc (cols * sizeof (int));
        ptr3[ i ] = (int *) malloc (cols * sizeof (int));
    }

    printf ("Enter values in first matrix \n");
    inputmat (ptr1, rows, cols);
    printf ("Enter values in second matrix \n");
    inputmat (ptr2, rows, cols);
    calcdiff (ptr1, ptr2, ptr3, rows, cols);
    printf ("Display difference of the two matrices \n");
    dispmat (ptr3, rows, cols);
}

 void inputmat (int *ptr1[MAXROWS], int m, int n)
 {
    int i, j;
    for (i = 0; i < m; i++)
    {
        for (j = 0; j < n; j++)
        {
            scanf ("%d", (*(ptr1 + i) + j));
        }
    }
    return;
 }

 void dispmat (int *ptr3[  MAXROWS ], int m, int n)
 {
    int i, j;
    for (i = 0; i < m; i++)
    {
        for (j = 0; j < n; j++)
        {
            printf ("%d ", *(*(ptr3 + i) + j));
        }
        printf("\n");
    }
    return;
}

void calcdiff (int *ptr1[ MAXROWS ], int *ptr2 [ MAXROWS ],
                    int *ptr3[MAXROWS], int m, int n)
{
    int i, j;
    for (i = 0; i < m; i++)
    {
        for (j = 0; j < n; j++)
        {
            *(*(ptr3 + i) + j) = *(*(ptr1 + i) + j) - *(*(ptr2 + i) + j);
        }
    }
    return;
}
```

OUTPUT

```
Enter no. of rows & columns
3 3
Enter values in first matrix
2  6  3
5  9  3
1  0  2
Enter values in second matrix
3  5  7
2  8  2
1  0  1
Display difference of the two matrices
-1  1 -4
 3  1  1
 0  0  1
```

In this program, *ptr1*, *ptr2*, *ptr3* are each defined as an array of pointers to integers. Each array has a maximum of MAXROWS elements. Since each element of ptr1, ptr2, ptr3 is a pointer, we must provide each pointer with enough memory for each row of integers. This can be done using the library function *malloc* included in *stdlib.h* header file as follows:

*ptr1[i] = (int *) malloc (cols * sizeof (int));*

This function reserves a block of memory whose size (in bytes) is equivalent to *cols * sizeof(int).* Since cols = 3, so 3 * 2 (size of int data type) i.e., 6 is allocated to each *ptr1[1], ptr1[2]* and *ptr1[3].* This *malloc* function returns a pointer of type *void.* This means that we can assign it to any type of pointer. In this case, the pointer is type-casted to an integer type and assigned to the pointer *ptr1[1], ptr1[2]* and *ptr1[3].* Now, each of *ptr1[1], ptr1[2]* and *ptr1[3]* points to the first byte of the memory allocated to the corresponding set of one-dimensional integer arrays of the original two-dimensional array.

The process of calculating and allocating memory at run time is known as *dynamic memory allocation.* The library routine *malloc* can be used for this purpose.

Instead of using conventional array notation, pointer notation has been used for accessing the address and value of corresponding array elements which has been explained to you in the previous section. The difference of the array elements within the function *calcdiff* is written as

**(*(ptr3 + i) + j) = *(*(ptr1 + i) + j) - *(*(ptr2 + i) + j);*

10.9 POINTERS AND STRINGS

As we have seen in strings, a string in C is an array of characters ending in the null character (written as '\0'), which specifies where the string terminates in memory. Like in one-dimensional arrays, a string can be accessed via a pointer to the first character in the string. The value of a string is the (constant) address of its first character. Thus, it is appropriate to say that a string is a constant pointer.
A string can be declared as a character array or a variable of type *char **. The declarations can be done as shown below:

```
char country[ ] = "INDIA";
char *country = "INDIA";
```

Each initialize a variable to the string “INDIA”. The second declaration creates a pointer variable *country* that points to the letter I in the string "INDIA" somewhere in memory.

Once the base address is obtained in the pointer variable *country*, **country* would yield the value at this address, which gets printed through,

```
printf ("%s", *country);
```

Here is a program that dynamically allocates memory to a character pointer using *the* library function *malloc* at run-time. An advantage of doing this way is that a fixed block of memory need not be reserved in advance, as is done when initializing a conventional character array.

Example 10.13

Write a program to test whether the given string is a palindrome or not.

/* Program tests a string for a palindrome using pointer notation */

```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>

main()
{
   char *palin, c;
   int i, count;

   short int palindrome(char,int);          /*Function Prototype */
   palin = (char *) malloc (20 * sizeof(char));
   printf("\nEnter a word: ");
   do
```

```
    {
         c = getchar( );
         palin[i] = c;
         i++;
    }while (c != '\n');

    i = i-1;
    palin[i] = '\0';
    count = i;

    if (palindrome(palin,count) == 1)
         printf ("\nEntered word is not a palindrome.");
    else
          printf ("\nEntered word is a palindrome");
    }

short int palindrome(char *palin, int len)
{
       short int i = 0, j = 0;
       for(i=0 , j=len-1; i < len/2;i++,j--)
       {
            if (palin[i] == palin[j])
                   continue;
            else
                   return(1);
        }
        return(0);
}
```

OUTPUT

```
Enter a word: malayalam

Entered word is a palindrome.

Enter a word: abcdba

Entered word is not a palindrome.
```

Array of pointers to strings

Arrays may contain pointers. We can form an array of strings, referred to as a string array. Each entry in the array is a string, but in C a string is essentially a pointer to its first character, so each entry in an array of strings is actually a pointer to the first character of a string. Consider the following declaration of a string array:

```
char *country[ ] = {
                        "INDIA", "CHINA", "BANGLADESH", "PAKISTAN", "U.S"
                          };
```

The **country[ ]* of the declaration indicates an array of five elements. The *char** of the declaration indicates that each element of array country is of type "pointer to char". Thus, *country [0]* will point to INDIA, *country[1]* will point to CHINA, and so on.

Thus, even though the array *country* is fixed in size, it provides access to character strings of any length. However, a specified amount of memory will have to be allocated for each string later in the program, for example,

```
country[ i ] = (char *) malloc(15 * sizeof (char));
```

The *country* character strings could have been placed into a two-dimensional array but such a data structure must have a fixed number of columns per row, and that number must be as large as the largest string. Therefore, considerable memory is wasted when a large number of strings are stored with most strings shorter than the longest string.

As individual strings can be accessed by referring to the corresponding array element, individual string elements be accessed through the use of the indirection operator. For example, ** (* country + 3) + 2)* refers to the third character in the fourth string of the array *country*. Let us see an example below.

Example 10.14

Write a program to enter a list of strings and rearrange them in alphabetical order, using a one-dimensional array of pointers, where each pointer indicates the beginning of a string:

```
/* Program to sort a list of strings in alphabetical order using an array of pointers */

# include <stdio.h>
# include <conio.h>
# include <stdlib.h>
# include <string.h>

void readinput (char *[ ], int);
void writeoutput (char *[ ], int);
void reorder (char *[ ], int);

main( )
{
       char *country[ 5 ];
       int i;
       for (i = 0; i < 5; i++)
           {
            country[ i ] = (char *) malloc (15 * sizeof (char));
           }
      printf ("Enter five countries on a separate line\n");
      readinput (country, 5);
      reorder (country, 5);
      printf ("\nReordered list\n");
      writeoutput (country, 5);
      getch( );
}

void readinput (char *country[ ], int n)
{
        int i;
        for (i = 0; i < n; i++)
                {   scanf ("%s", country[ i ]);   }
          return;
}

void writeoutput (char *country[ ], int n)
{
         int i;
        for (i = 0; i < n; i++)
        {   printf ("%s", country[ i ]);
            printf ("\n");   }
         return;
}

void reorder (char *country[ ], int n)
{
         int i, j;
         char *temp;
         for (i = 0; i < n-1; i++)
         {
             for (j = i+1; j < n; j++)
             {
                 if (strcmp (country[ i ], country[ j ]) > 0)
                 {
                      temp = country[ i ];
                      country[ i ] = country[ j ];
                      country[ j ] = temp;
                  }
              }
           }
           return;
 }
```

OUTPUT

```
Enter five countries on a seperate line
INDIA
BANGLADESH
PAKISTAN
CHINA
SRILANKA

Reordered list
BANGLADESH
CHINA
INDIA
PAKISTAN
SRILANKA
```

The limitation of the string array concept is that when we are using an array of pointers to strings we can initialize the strings at the place where we are declaring the array, but we cannot receive the strings from keyboard using *scanf( ).*

Check Your Progress 3

1. What is meant by array of pointers?

…………………………………………………………………………………………

…………………………………………………………………………………………

2. How the indirection operator can be used to access a multidimensional array element.

…………………………………………………………………………………………

…………………………………………………………………………………………

3. A C program contains the following declaration.
   float temp[3][2] = {{13.4, 45.5}, {16.6, 47.8}, {20.2, 40.8}};

   (i) What is the meaning of temp?
   (ii) What is the meaning of (temp + 2)?
   (iii) What is the meaning of *(temp + 1)?
   (iv) What is the meaning of (*(temp + 2) + 1)?
   (v) What is the meaning of *(*(temp) + 1) + 1)?
   (vi) What is the meaning of *(*(temp + 2))?

   ……………………………………………………………………………………

   ……………………………………………………………………………………

   ……………………………………………………………………………………

10.10 SUMMARY

In this unit we have studied about pointers, pointer arithmetic, passing pointers to functions, relation to arrays and the concept of dynamic memory allocation. A pointer is simply a variable that contains an address which is a location of another variable in memory. The unary operator &, when preceded by any variable returns its address. C's other unary pointer operator is *, when preceded by a pointer variable returns a value stored at that address.

Pointers are often passed to a function as arguments by reference. This allows data items within the calling function to be accessed, altered by the called function, and then returned to the calling function in the altered form. There is an intimate relationship between pointers and arrays as an array name is really a pointer to the first element in the array. Access to the elements of array using pointers is enabled by adding the respective subscript to the pointer value (i.e. address of zeroth element) and the expression preceded with an indirection operator.

As pointer declaration does not allocate memory to store the objects it points at, therefore, memory is allocated at run time known as *dynamic memory allocation.* The library routine *malloc* can be used for this purpose.

10.11 SOLUTIONS / ANSWERS

Check Your Progress 1

1. Refer to section 10.4. The data type included in the pointer declaration, refers to the type of data stored at the address which we will be storing in our pointer.

2. (i) Compile-time Error : Lvalue Required. Means that the left side of an assignment operator must be an addressable expression that include a variable or an indirection through a pointer.

   (ii) Multiplication of a pointer variable with a constant is invalid.

3. (i) Refer section 10.4
   (ii) Refer section 10.4
   (iii) This means pointers can be of type void but can't be de-referenced without explicit casting. This is because the compiler can't determine the size of the object the pointer points to.

Check Your Progress 2

1 (i) True.
(ii) True.
(iii) False.
(iv) True.
(v) True.
(vi) True.

Check Your Progress 3

1. Refer section 10.4.

2. Refer section 10.4 to comprehend the convention followed.

3. (i) Refers to the base address of the array temp.

   (ii) Address of the first element of the last row of array temp i.e. address of element 20.2.

   (iii) Will give you 0. To get the value of the last element of the first array i.e. the correct syntax would be *(*(temp+0)+1).

   (iv) Address of the last element of last row of array temp i.e. of 40.8.

   (v) Displays the value 47.8 i.e., second element of last row of array temp.

   (vi) Displays the value 20.2 i.e., first element of last row of array temp.

10.12 FURTHER READINGS

1. Programming with C, Second Edition, *Gottfried Byron S*, Tata McGraw Hill, India.
2. The C Programming Language, Second Edition, *Brian Kernighan and Dennis Richie*, PHI, 2002.
3. Programming in ANSI C, Second Edition, *Balaguruswamy E*, Tata McGraw Hill, India, 2002.
4. How to Solve it by Computer, *R.G.Dromey*, PHI, 2002.
5. C Programming in 12 easy lessons, *Greg Perry*, SAMS, 2002.
6. Teach Yourself C in 21 days, Fifth Edition, *Peter G*, Fifth edition,SAMS, 2002.

UNIT 11 THE C PREPROCESSOR

Structure

11.0 INTRODUCTION

Theoretically, the "preprocessor" is a translation phase that is applied to the source code before the compiler gets its hands on it. The C Preprocessor is not part of the compiler, but is a separate step in the compilation process. C Preprocessor is just a text substitution tool, which filters your source code before it is compiled. The preprocessor more or less provides its own language, which can be a very powerful tool for the programmer. All preprocessor directives or commands begin with the symbol #.

The preprocessor makes programs easier to develop, read and modify. The preprocessor makes C code portable between different machine architectures & customizes the language.

The preprocessor performs textual substitutions on your source code in three ways:

File inclusion: Inserting the contents of another file into your source file, as if you had typed it all in there.
Macro substitution: Replacing instances of one piece of text with another.
Conditional compilation: Arranging that, depending on various circumstances, certain parts of your source code are seen or not seen by the compiler at all.

The next three sections will introduce these three preprocessing functions. The syntax of the preprocessor is different from the syntax of the rest of C program in several respects. The C preprocessor is not restricted to use with C programs, and programmers who use other languages may also find it useful. However, it is tuned to recognize features of the C language like comments and strings.

11.1 OBJECTIVES

After going through this unit, you will be able to:

- define, declare preprocessor directives;
- discuss various preprocessing directives, for example file inclusion, macro substitution, and conditional compilation; and
- discuss various syntaxes of preprocessor directives and their applications.

11.2 # *define* TO IMPLEMENT CONSTANTS

The preprocessor allows us to customize the language. For example to replace **{** and **}** of C language to *begin* and *end* as block-statement delimiters (as like the case in PASCAL) we can achieve this by writing:

```
# define begin {
# define end   }
```

During compilation all occurrences of *begin* and *end* get replaced by corresponding **{** and **}**. So the subsequent C compilation stage does not know any difference!

#define is used to define constants.

The syntax is as follows:

# define <literal> <replacement-value>

literal is identifier which is replaced with *replacement-value* in the program.

For Example,

```
#define MAXSIZE 256
#define PI 3.142857
```

The C preprocessor simply searches through the C code before it is compiled and replaces every instance of *MAXSIZE* with 256.

```
# define  FALSE  0
# define  TRUE   !FALSE
```

The literal *TRUE* is substituted by *!FALSE* and *FALSE* is substituted by the value 0 at every occurrence, before compilation of the program. Since the values of the literal are constant throughout the program, they are called as constant.

The syntax of above # *define* can be rewritten as:

# define <constant-name> <replacement-value>

Let us consider some examples,

```
# define      M           5
# define      SUBJECTS    6
# define      PI          3.142857
# define      COUNTRY     INDIA
```

Note that no semicolon (;) need to be placed as the delimiter at the end of a # define line. This is just one of the ways that the syntax of the preprocessor is different from the rest of C statements (commands). If you unintentionally place the semicolon at the end as below:

```
#define MAXLINE 100;          /* WRONG */
```

and if you declare as shown below in the declaration section,

```
char line[MAXLINE];
```

the preprocessor will expand it to:

```
char line[100;];                         /* WRONG */
```

which gives you the syntax error. This shows that the preprocessor doesn't know much of anything about the syntax of C.

11.3 # *define* TO CREATE FUNCTIONAL MACROS

Macros are inline code, which are substituted at compile time. The definition of a macro is that which accepts an argument when referenced. Let us consider an example as shown below:

Example 11.1

Write a program to find the square of a given number using macro.

```
/* Program to find the square of a number using marco*/
#include <stdio.h>
# define  SQUARE(x)   (x*x)
main()
   {
    int v,y;
    printf("Enter any number to find its square: ");
    scanf("%d", &v);
    y = SQUARE(v);
    printf("\nThe square of %d is %d", v, y);
}
```

OUTPUT

```
Enter any number to find its square: 10
The square of 10 is 100
```

In this case, *v* is equated with *x* in the macro definition of *square*, so the variable *y* is assigned the square of *v*. The brackets in the macro definition of *square* are necessary for correct evaluation. The expansion of the macro becomes:

*y =(v * v);*

Macros can make long, ungainly pieces of code into short words. Macros can also accept parameters and return values. Macros that do so are called *macro functions*. To create a macro, simply define a macro with a parameter that has whatever name you like, such as *my_val*. For example, one macro defined in the standard libraries is "abs", which returns the absolute value of its parameter. Let us define our own version of ***ABS*** as shown below. Note that we are defining it in upper case not only to avoid conflicting with the existing "abs".

```
#define ABS(my_val) ((my_val) < 0) ? -(my_val) : (my_val)
```

#define can also be given arguments which are used in its replacement. The definitions are then called macros. Macros work rather like functions, but with the following minor differences:

- Since macros are implemented as a textual substitution, by this the performance of program improves compared to functions.
- Recursive macros are generally not a good idea.

- Macros don't care about the type of their arguments. Hence macros are a good choice where we want to operate on reals, integers or a mixture of the two. Programmers sometimes call such type flexibility polymorphism.
- Macros are generally fairly small.

Let us look more illustrative examples to understand the *macros* concept.

Example 11.2

Write a program to declare constants and macro functions using *#define*.

```
/* Program to illustrate the macros */
#include <stdio.h>
#include <string.h>
#define STR1        "A macro definition!\n"
#define STR2        "must be all on one line!\n"
#define EXPR1        1+2+3
#define EXPR2        EXPR1+5
#define ABS(x)       (((x) < 0) ? – (x):(x))
#define MAX(p,q)    ((p < q) ? (q):(p))
#define BIGGEST(p,q,r)   (MAX(p, q) < r)?(r):(MAX(p, q))
main()
{
  printf(STR1);
  printf(STR2);
  printf("Largest number among  %d, %d and %d is %d\n",EXPR1, EXPR2, ABS (–3),
                                                         BIGGEST(1,2,3));
}
```

OUTPUT

```
A macro definition
must be all on one line!
Largest number among 6, 11 and 3 is 3
```

The macro STR1 is replaced with "A macro definition \n" in the first *printf()* function. The macro STR2 is replaced with "must be all on one line! \n" in the second *printf* function. The macro EXPR1 is replaced with 1+2+3 in third *printf* statement. The macro EXPR2 is replaced with EXPR1 +5 in fourth *printf* statement. The macro ABS(–3) is replaced with (– 3<0) ? – (– 3) : 3. And evaluation 3 is replaced. The largest among the three numbers is diplayed.

Example 11.3

Write a program to find out square and cube of any given number using macros.

```
/* Program to find  the square and cube of any given number using macro directive */
# include<stdio.h>
# define   sqr(x)            (x  * x)
# define   cub(x)            (sqr(x) *  x)
main()
{
        int  num;
        printf("Enter a number: ");
        scanf("%d", &num);
        printf(" \n Square of  the number is %d", sqr(num));
        printf(" \n Cube of the number is %d\n", cub(num));
}
```

OUTPUT

```
Enter a number: 5
Square of the number is 25
Cube of the number is 125
```

Note: Multi-line macros can be defined by placing a backward slash (\ ) at end of each line except the last line. This feature permits a single macro (i.e. a single identifier) to represent a compound statement.

Example 11.4

Write a macro to display the string COBOL in the following fashion

```
C
CO
COB
COBO
COBOL
COBOL
COBO
COB
CO
C
```

```
/* Program to display the string as given in the problem*/
# include<stdio.h>
# define        LOOP    for(x=0; x<5; x++)             \
                   {     y=x+1;                        \
                        printf("%-5.*s\n", y, string);  }   \
                       for(x=4; x>=0; x--)             \
                        {          y=x+1;              \
                        printf("%-5.*s \n", y, string); }

main()
{
        int x, y;
        static char string[ ] = "COBOL";
        printf("\n");
        LOOP;
}
```

When the above program is executed the reference to macro (loop) is replaced by the set of statements contained within the macro definition.

OUTPUT

```
C
CO
COB
COBO
COBOL
COBOL
COBO
COB
CO
C
```

Recollect that CALL BY VALUE Vs CALL BY REFERENCE given in the previous uint. By CALL BY VALUE, the swapping was not taking place, because the visibility of the variables was restricted to with in the function in the case of local variables. You can resolve this by using a macro. Here is ***swap*** in action when using a macro:

#define swap(x, y) {int tmp = x; x = y; y = tmp; }

Now we have swapping code that works. Why does this work? It is because the CPP just simply replaces text. Wherever swap is called, the CPP will replace the macro call with the macro meaning, (defined text).

Caution in using macros

You should be very careful in using Macros. In particular the textual substitution means that arithmetic expressions are liable to be corrupted by the order of evaluation rules (precedence rules). Here is an example of a macro, which won't work.

```
#define     DOUBLE(n)           n + n
```

Now if we have a statement,

```
z = DOUBLE(p) *  q;
```

This will be expanded to

```
z = p + p * q;
```

And since * has a higher priority than +, the compiler will treat it as.

```
z = p + (p * q);
```

The problem can be solved using a more robust definition of DOUBLE

```
#define     DOUBLE(n)           (n + n)
```

Here, the braces around the definition force the expression to be evaluated before any surrounding operators are applied. This should make the macro more reliable.

Check Your Progress 1

1. Write a macro to evaluate the formula *f(x) = x*x + 2*x + 4.*

……………………………………………………………………………………..……………

………………………………………………………………………………………………..

2. Define a preprocessor macro *swap(t, x, y)* that will swap two arguments *x* and *y* of a given type *t*.

………………………………………………………………………………………………

………………………………………………………………………………………………

3. Define a macro called *AREA*, which will calculate the area of a circle in terms of radius.

………………………………………………………………………………………………

………………………………………………………………………………………………

4. Define a macro called *CIRCUMFERENCE*, which will calculate the circumference of a circle in terms of radius.

………………………………………………………………………………………

………………………………………………………………………………………

5. Define a macro to display multiplication table.

………………………………………………………………………………………

………………………………………………………………………………………

6. Define a macro to find sum of *n* numbers.

………………………………………………………………………………………

………………………………………………………………………………………

……………………………………………………………………………………….

11.4 READING FROM OTHER FILES USING # *include*

The preprocessor directive *#include* is an instruction to read in the entire contents of another file at that point. This is generally used to read in header files for library functions. Header files contain details of functions and types used within the library. They must be included before the program can make use of the library functions. The syntax is as follows:

#include <filename.h>

or

#include "*filename.h*"

The above instruction causes the contents of the file "filename.h" to be read, parsed, and compiled at that point. The difference between the suing of # and " " is that, where the preprocessor searches for the *filename.h.* For the files enclosed in **< >** (less than and greater than symbols) the search will be done in standard directories (include directory) where the libraries are stored. And in case of files enclosed in " " (double quotes) search will be done in "current directory" or the directory containing the source file. Therefore, " " is normally used for header files you've written, and # is normally used for headers which are provided for you (which someone else has written).

Library header file names are enclosed in angle brackets, < >. These tell the preprocessor to look for the header file in the standard location for library definitions. This is ***/usr/include*** for most UNIX systems. And ***c:/tc/include*** for turbo compilers on DOS / WINDOWS based systems.

Use of *#include* for the programmer in multi-file programs, where certain information is required at the beginning of each program file. This can be put into a file by name "globals.h" and included in each program file by the following line:

#include "globals.h"

If we want to make use of inbuilt functions related to input and output operations, no need to write the prototype and definition of the functions. We can simply include the file by writing:

#include <stdio.h>

and call the functions by the function calls. The standard header file *stdio.h* is a collection of function prototype (declarations) and definition related to input and output operations.

The extension "*.h*"', simply stands for "header" and reflects the fact that *#include* directives usually sit at the top (head) of your source files. "*.h*" extension is not compulsory – you can name your own header files anything you wish to, but *.h* is traditional, and is recommended.

Placing common declarations and definitions into header files means that if they always change, they only have to be changed in one place, which is a much more feasible system.

What should you put in header files?

- External declarations of global variables and functions.
- Structure definitions.
- Typedef declarations.

However, there are a few things *not* to put in header files:

- Defining instances of global variables. If you put these in a header file, and include the header file in more than one source file, the variable will end up multiply defined.
- Function bodies (which are also defining instances), may not be put in header files. Since these headers may end you up with multiple copies of the function and hence "multiply defined" errors. People sometimes put commonly-used functions in header files and then use #include to bring them (once) into each program where they use that function, or use #include to bring together the several source files making up a program, but both of these are not good practice. It's much better to learn how to use your compiler or linker to combine together separately-compiled object files.

11.5 CONDITIONAL SELECTION OF CODE USING # *ifdef*

The preprocessor has a conditional statement similar to C's if-else. It can be used to selectively include statements in a program. The commands for conditional selection are; *#ifdef, #else* and *#endif.*

#ifdef

The syntax is as follows:

```
#ifdef  IDENTIFIER_NAME
{
 statements;
}
```

This will accept a name as an argument, and returns true if the name has a current definition. The name may be defined using a # *define*, the *-d* option of the compiler, or certain names which are automatically defined by the UNIX environment. If the identifier is defined then the statements below #ifdef will be executed

#else

The syntax is as follows:

```
#else
{
statements;
```

}

#else is optional and ends the block started with *#ifdef*. It is used to create a 2 way optional selection. If the identifier is not defined then the statements below *#else* will be executed.

#endif

Ends the block started by *#ifdef* or *#else*.

Where the *#ifdef* is true, statements between it and a following *#else* or *#endif* are included in the program. Where it is false, and there is a following *#else*, statements between the *#else* and the following *#endif* are included. Let us look into the illustrative example given below to get an idea.

Example 11.5

Define a macro to find maximum of 3 or 2 numbers using #ifdef , #else

```
/* Program to find maximum of 2 numbers using #ifdef*/

#include<stdio.h>
#define TWO
main()
{
int a, b, c;
clrscr();

#ifdef   TWO
  {
  printf("\n Enter two numbers: \n");
  scanf("%d %d", &a,&b);
  if(a>b)
        printf("\n Maximum  of two numbers is %d", a);
    else
        printf("\n Maximum  is of two numbers is %d", b);
  }

#endif
}   /* end of main*/
```

OUTPUT

```
Enter two numbers:
33
22
Maximum  of two numbers is  33
```

Explanation

The above program demonstrate preprocessor derivative #ifdef. By using #ifdef TWO has been defined. The program finds out the maximum of two numbers.

11.5.1 Using #ifdef for Different Computer Types

Conditional selection is rarely performed using *#define* values. This is often used where two different computer types implement a feature in different ways. It allows the programmer to produce a program, which will run on either type.

A simple application using machine dependent values is illustrated below.

```
#include <stdio.h>
main()
{
#ifdef HP
{
    printf("This is a HP system \n");
    …………………….
    ……………………. /* code for HP  systems*/
      }
 #endif

#ifdef SUN
{
        printf("This is a SUN system \n");
        …………………….         /* code for SUN Systems
}
#endif
}
```

If we want the program to run on HP systems, we include the directive
#define HP at the top of the program.

If we want the program to run on SUN systems, we include the directive
#define SUN at the top of the program.

Since all you're using the macro HP or SUN to control the #ifdef, you don't need to give it *any replacement* text at all. *Any* definition for a macro, even if the replacement text is empty, causes an #ifdef to succeed.

11.5.2 Using #ifdef to Temporarily Remove Program Statements

#ifdef also provides a useful means of temporarily "blanking out" lines of a program. The lines in the program are preceded by *#ifdef* NEVER and followed by *#endif*. Of course, you should ensure that the name NEVER isn't defined anywhere.

```
#include <stdio.h>
main()
{
…………….
#ifdef NEVER
{
    …………………….
    ……………………. /* code is skipped */
   #endif
}
```

11.6 OTHER PREPROCESSOR COMMANDS

Other preprocessor commands are:

- **#ifndef** - If this macro is not defined
- **#if** - Test if a compile time condition is true

- **#else** - The alternative for #if. This is part of an #if preprocessor statement and works in the same way with #if that the regular C else does with the regular if.
- **#elif** - enables us to establish an "if…else…if .." sequence for testing multiple conditions.

Example 11.6

```
#if  processor  == intel
#define FILE "intel.h"
#elif  processor  == amd
#define FILE "amd.h"
#if  processor  == motrola
#define FILE "motrola.h"
#endif
#include FILE
```

- **#** - Stringizing operator, to be used in the definition of macro. This operator allows a formal parameter within macro definition to be converted to a string.

Example 11.7

```
#define multiply (p*q) printf(#pq " = %f", pq)
main()
{
   ………..
   multiply(m*n);
}
```

The preprocessor converts the line *multiply(m*n)* into *printf("m*n" " = %f", m*n);* And then into printf("m*n = %f", m*n);

- Token merge, creates a single token from two adjacent ones within a macro definition.

Example 11.8

```
#define merge(s1,s2)  s1## s2
main()
{
   …………..
   printf("%f", merge(total, sales);
}
```

The preprocessor transforms the statement *merge(total, sales) into printf("%f", totalsales);*

#error - text of error message -- generates an appropriate compiler error message.

Example 11.9

```
#ifdef  OS_MSDOS
#include <msdos.h>
      #elifdef OS_UNIX
          #include ``default.h"
              #else
```

```
                #error Wrong OS!!
#endif
```

# line

#line number *"string"* – informs the preprocessor that the number is the next number of line of input. *"string"* is optional and names the next line of input. This is most often used with programs that translate other languages to C. For example, error messages produced by the C compiler can reference the file name and line numbers of the original source files instead of the intermediate C (translated) source files.

#pragma

It is used to turn on or off certain features. Pragmas vary from compiler to compiler. Pragmas available with Microsoft C compilers deals with formatting source listing and placing comments in the object file generated by the compiler. Pragmas available with Turbo C compilers allows to write assembly language statements in C program.

A control line of the form

#pragma   *token-sequence*

This causes the processor to perform an implementation-dependent action. An unrecognized pragma is ignored.

Other preprocessor directives are # - Stringizing operator allows a formal parameter within macro definition to be converted to a string. **## -** Token merge, creates a single token from two adjacent ones within a macro definition. **#error -** generates an appropriate compiler error message.

Example 11.10

Write a macro to demonstrate #define, #if, #else preprocessor commands.

```
/* The following code displays 35 to the screen.  */

#include <stdio.h>
#define CHOICE 100
int my_int = 0;
#if (CHOICE == 100)
        void set_my_int()
         {    my_int = 35;        }
#else
         void set_my_int()
         {
              my_int = 27;
          }
#endif
main ()
{
              set_my_int();
              printf("%d\n", my_int);
}
```

OUTPUT

```
35
```

The *my_int* is initialized to zero and *CHOICE* is defined as 100. *#if* derivative checks whether *CHOICE* is equal to 100. Since *CHOICE* is defined as 100, *void set_my_int* is called and *int* is set 35. And the same is displayed on to the screen.

Example 11.11

Write a macro to demonstrate #define, #if, #else preprocessor commands.

```
/*  The following code displays 27 on the screen */

#include <stdio.h>
#define CHOICE 100
int my_int = 0;
#undef CHOICE
#ifdef CHOICE
        void set_my_int()
        {
                my_int = 35;
        }
#else
        void set_my_int()
        {
                my_int = 27;
        }
#endif

main ()
{
        set_my_int();
        printf("%d\n", my_int);
}
```

OUTPUT

```
27
```

The *my_int* is initialized to 0 and *CHOICE* is defined as 100. *#undef* is used to undefine CHOICE. *#else* is invoked , *void set_my_int* is called and *int* is set 27. And the same is displayed on to the screen.

11.7 PREDEFINED NAMES DEFINED BY PREPROCESSOR

These are identifiers defined by the preprocessor, and cannot be undefined or redefined. They are:

LINE	an integer constant containing the current source line number.
FILE	a string containing the name of the file being complied.
DATE	a string literal containing the date of compilation, in the form "mm-dd-yyyy".
TIME	a string literal containing the time of compilation, in the form "hh:mm:ss".
STDC	the constant 1. This identifier is defined to be 1 only in the implementations conforming to the ANSI standard.

11.8 MACROS Vs FUNCTIONS

Till now we have discussed about macros. Any computations that can be done on macros can also be done on functions. But there is a difference in implementations and in some cases it will be appropriate to use macros than function and vice versa. We will see the difference between a macro and a function now.

Macros	**Functions**
Macro calls are replaced with macro expansions (meaning).	In function call, the control is passed to a function definition along with arguments, and definition is processed and value may be returned to call
Macros run programs faster but increase the program size.	Functions make program size smaller and compact.
If macro is called 100 numbers of times, the size of the program will increase.	If function is called 100 numbers of times, the program size will not increase.
It is better to use Macros, when the definition is very small in size.	It is better to use functions, when the definition is bigger in size.

Check Your Progress 2

1. Write an instruction to the preprocessor to include the math library `math.h`.

…………………………………………………………………………………………

…………………………………………………………………………………………

2. Write a macro to add user defined header file by name *madan.h* to your program.

…………………………………………………………………………………………

…………………………………………………………………………………………

3. What will be the output of the following program?

```
#include<stdio.h>
main()
{
    float  m=7;
  #ifdef DEF
    i*=i;
  #else
    printf("\n%f", m);
  #endif      }
```

…………………………………………………………………………………………

…………………………………………………………………………………………

4. Write a macro to find out whether the given character is upper case or not.

…………………………………………………………………………………………

…………………………………………………………………………………………

…………………………………………………………………………………………..

11.9 SUMMARY

The preprocessor makes programs easier to develop and modify. The preprocessor makes C code more portable between different machine architectures and customize the language. The C Preprocessor is not part of the compiler, but is a separate step in the compilation process. All preprocessor lines begin with #. C Preprocessor is just a text substitution tool on your source code in three ways: File inclusion, Macro substitution, and Conditional compilation.

File inclusion - inserts the contents of another file into your source file.
Macro Substitution - replaces instances of one piece of text with another.
Conditional Compilation - arranges source code depending on various circumstances.

11.10 SOLUTIONS / ANSWERS

Check Your Progress 1

1.
```
# include<stdio.h>
# define f(x)       x*x + 2 * x + 4
main()
{
int num;
printf("enter value x: ");
scanf("%d",&num);
printf("\nvalue of f(num) is %d", f(num));
}
```

2.
```
# include<stdio.h>
# define swap(t, x, y)  { t  tmp = x; x = y; y = tmp; }
main( )
{
 int      a, b;
 float   p, q;
printf("enter integer values for a, b: ");
scanf("%d %d", &a, &b);
printf("\n Enter float values for p, q: ");
scanf("%f %f", &p, &q);
swap(int, a, b);
printf(" \n After swap the values of  a and b are %d %d", a, b);
swap(float, p, q);
printf("\n After swap the values of p and q are %f %f", p, q);
}
```

3.
```
# include<stdio.h>
# define   AREA(radius)      3.1415 * radius * radius
 main( )
 {
     int radius;
     printf("Enter value of radius:  ");
     scanf("%d ", &radius );
     printf("\nArea is  %d", AREA(radius));
 }
```

4.
```
# include<stdio.h>
# define   CIRCUMFERENCE(radius)        2 * 3.1415 * radius

 main( )
```

```
{
    int radius;
    printf("Enter value for radius ");
    scanf("%d ", &radius );
    printf("Circumference is  %d", CIRCUMFERENCE(radius));
}
```

5.
```
# include<stdio.h>
# define MUL_TABLE(num)          for(n=1;n<=10;n++)   \
                              printf("\n%d*%d=%d",num,n,num*n)
main()
{
    int number;     int n;
    printf("enter number");
    scanf("%d",&number);
    MUL_TABLE(number);
 }
```

6.
```
# include<stdio.h>
# define SUM(n)  ( (n * (n+1)) / 2 )
main()
{
    int number;
    printf("enter number");
    scanf("%d", &number);
    printf("\n sum of n numbers %d", sum(number));      }
```

Check Your Progress 2

1. # include <math.h>

2. #include "madan.h"

3. 7

4.
```
# include<stdio.h>
# define isupper(c) (c>=65 && c<=90)
main()
{
  char c;
  printf("Enter character:");
  scanf("%c",&c);
  if(isupper(c))
       printf("\nUpper case");
    else
         printf("\nNo it is not an upper case character");
}
```

11.11 FURTHER READINGS

1. The C programming language, *Brian W. Kernighan & Dennis Ritchie*, Pearson Education, 2002.
2. Programming with ANSI and Turbo C, *Ashok N. Kamthane*, Pearson Education, 2002.
3. Computer Programming in C, *Raja Raman. V*, PHI, 2002.

UNIT 12 FILES

Structure

12.0 INTRODUCTION

The examples we have seen so far in the previous units deal with standard input and output. When data is stored using variables, the data is lost when the program exits unless something is done to save it. This unit discusses methods of working with files, and a data structure to store data. C views file simply as a sequential stream of bytes. Each file ends either with an *end-of-file* marker or at a specified byte number recorded in a system maintained, administrative data structure. C supports two types of files called ***binary*** *files* and ***text*** *files*.

The difference between these two files is in terms of storage. In *text files*, everything is stored in terms of text *i.e.* even if we store an integer 54; it will be stored as a 3-byte string - "54\0". In a text file certain character translations may occur. For example a *newline(\n)* character may be converted to a carriage return, linefeed pair. This is what Turbo C does. Therefore, there may not be one to one relationship between the characters that are read or written and those in the external device. A *binary file* contains data that was written in the same format used to store internally in main memory.

For example, the integer value 1245 will be stored in 2 bytes depending on the machine while it will require 5 bytes in a text file. The fact that a numeric value is in a standard length makes binary files easier to handle. No special string to numeric conversions is necessary.

The disk I/O in C is accomplished through the use of library functions. The ANSI standard, which is followed by TURBO C, defines one complete set of I/O functions. But since originally C was written for the UNIX operating system, UNIX standard defines a second system of routines that handles I/O operations. The first method, defined by both standards, is called a buffered file system. The second is the unbuffered file system.

In this unit, we will first discuss buffered file functions and then the unbuffered file functions in the following sections.

12.1 OBJECTIVES

After going through this unit you will be able to:

- define the concept of file pointer and file storage in C;
- create text and binary files in C;
- read and write from text and binary files;
- deal with large set of Data such as File of Records; and
- perform operations on files such as count number of words in a file, search a word in a file, compare two files etc.

12.2 FILE HANDLING IN C USING FILE POINTERS

As already mentioned in the above section, a sequential stream of bytes ending with an *end-of-file* marker is what is called a ***file***. When the file is opened the stream is associated with the file. By default, three files and their streams are automatically opened when program execution begins - the ***standard input***, ***standard output***, and the ***standard error***. Streams provide communication channels between files and programs.

For example, the standard input stream enables a program to read data from the keyboard, and the standard output stream enables to write data on the screen. Opening a file returns a pointer to a FILE structure (defined in <stdio.h>) that contains information, such as size, current file pointer position, type of file etc., to perform operations on the file. This structure also contains an integer called a *file descriptor* which is an index into the table maintained by the operating system namely, the *open file table.* Each element of this table contains a block called *file control block (FCB)* used by the operating system to administer a particular file.

The standard input, standard output and the standard error are manipulated using file pointers *stdin, stdout* and *stderr.* The set of functions which we are now going to discuss come under the category of buffered file system. This file system is referred to as buffered because, the routines maintain all the disk buffers required for reading / writing automatically.

To access any file, we need to declare a pointer to FILE structure and then associate it with the particular file. This pointer is referred as a *file pointer* and it is declared as follows:

*FILE *fp;*

12.2.1 Open A File Using The Function *fopen()*

Once a file pointer variables has been declared, the next step is to open a file. The *fopen()* function opens a stream for use and links a file with that stream. This function returns a file pointer, described in the previous section. The syntax is as follows:

*FILE *fopen(char *filename,*mode);*

where ***mode*** is a string, containing the desired open status. The filename must be a string of characters that provide a valid file name for the operating system and may include a path specification. The legal mode strings are shown below in the table 12.1:

Table 12.1: Legal values to the *fopen( )* mode parameter

MODE	MEANING
"r" / "rt"	opens a text file for read only access
"w" / "wt"	creates a text file for write only access
"a" / "at"	text file for appending to a file
"r+t"	open a text file for read and write access
"w+t"	creates a text file for read and write access,
"a+t"	opens or creates a text file and read access
"rb"	opens a binary file for read only access
"wb"	create a binary file for write only access
"ab"	binary file for appending to a file
"r+b"	opens a binary or read and write access
"w+b"	creates a binary or read and write access,
"a+b"	open or binary file and read access

The following code fragment explains how to open a file for reading.

Code Fragment 1

```
#include <stdio.h>

main ()
  {

    FILE *fp;
    if ((fp=fopen("file1.dat", "r"))==NULL)
    {
   printf("FILE DOES NOT EXIST\n");
   exit(0);
    }
 }
```

The value returned by the *fopen( )* function is a file pointer. If any error occurs while opening the file, the value of this pointer is *NULL*, a constant declared in <*stdio.h*>. Always check for this possibility as shown in the above example.

12.2.2 Close A File Using The Function Fclose( )

When the processing of the file is finished, the file should be closed using the fclose() function, whose syntax is:

*int fclose(FILE *fptr);*

This function flushes any unwritten data for stream, discards any unread buffered input, frees any automatically allocated buffer, and then closes the stream. The return value is 0 if the file is closed successfully or a constant *EOF*, an end-of file marker, if an error occurred. This constant is also defined in <*stdio.h*>. If the function *fclose()* is not called explicitly, the operating system normally will close the file when the program execution terminates.

The following code fragment explains how to close a file.

Code Fragment 2

```
# include <stdio.h>
main ( )
{
 FILE *fp;
 if ((fp=fopen(“file1.dat”, “r”))==NULL)
    {
      printf(“FILE DOES NOT EXIST\n”);
      exit(0);
    }
……………..
……………..
……………..
…………….
/* close the file */
fclose(fp);
}
```

Once the file is closed, it cannot be used further. If required it can be opened in same or another mode.

Check Your Progress 1

1. How does fopen( ) function links a file to a stream?

…………………………………………………………………………………………
…………………………………………………………………………………………
……………………………………………………………………………………..….

2. Differentiate between text files and binary files.

…………………………………………………………………………………………
…………………………………………………………………………………………
………………………………………………………………………………………..

3. What is EOF and what is its value?

…………………………………………………………………………………………
…………………………………………………………………………………………
……………………………………………………………………………………..….

12.3 INPUT AND OUTPUT USING FILE POINTERS

After opening the file, the next thing needed is the way to read or write the file. There are several functions and macros defined in <*stdio.h*> header file for reading and writing the file. These functions can be categorized according to the form and type of data read or written on to a file. These functions are classified as:

- Character input/output functions
- String input/output functions
- Formatted input/output functions
- Block input/output functions.

12.3.1 Character Input and Output in Files

ANSI C provides a set of functions for reading and writing character by character or one byte at a time. These functions are defined in the standard library. They are listed and described below:

- getc()
- putc()

getc( ) is used to read a character from a file and *putc( )* is used to write a character to a file. Their syntax is as follows:

*int putc(int ch, FILE *stream);*
*int getc(FILE *stream);*

The file pointer indicates the file to read from or write to. The character ***ch*** is formally called an integer in *putc( )* function but only the low order byte is used. On success *putc( )* returns a character(in integer form) written or EOF on failure. Similarly getc( ) returns an integer but only the low order byte is used. It returns *EOF* when end-of-file is reached. *getc( )* and *putc( )* are defined in <stdio.h> as macros not functions.

fgetc() and fputc()

Apart from the above two macros, C also defines equivalent functions to read / write characters from / to a file. These are:

*int fgetc(FILE *stream);*
*int fputc(int c, FILE *stream);*

To check the end of file, C includes the function *feof( )* whose prototype is:

*int feof(FILE *fp);*

It returns **1** if end of file has been reached or **0** if not. The following code fragment explains the use of these functions.

Example 12.1

Write a program to copy one file to another.

```
/*Program to copy one file to another */

 #include <stdio.h>
main( )
{
  FILE *fp1;
  FILE *fp2;
  int ch;
  if((fp1=fopen("f1.dat","r")) == NULL)

   {
   printf("Error opening input file\n");
    exit(0);
   }
      if((fp2=fopen("f2.dat","w")) == NULL)
  {
   printf("Error opening output file\n");
        exit(0);
        }

        while (!feof(fp1))
        {
          ch=getc(fp1);
          putc(ch,fp2);
        }
        fclose(fp1);
        fclose(fp2);
}
```

OUTPUT

If the file ”f1.dat” is not present, then the output would be:

```
            Error opening input file
```

If the disk is full, then the output would be:

```
            Error opening output file
```

If there is no error, then “f2.dat” would contain whatever is present in “f1.dat” after the execution of the program, if “f2.dat” was not empty earlier, then its contents would be overwritten.

12.3.2 String Input/Output Functions

If we want to read a whole line in the file then each time we will need to call character input function, instead C provides some string input/output functions with the help of which we can read/write a set of characters at one time. These are defined in the standard library and are discussed below:

- *fgets( )*
- *fputs( )*

These functions are used to read and write strings. Their syntax is:

*int fputs(char *str, FILE *stream);*
*char *fgets(char *str, int num, FILE *stream);*

The integer parameter in *fgets( )* is used to indicate that at most num-1 characters are to be read, terminating at end-of-file or end-of-line. The end-of-line character will be placed in the string ***str*** before the string terminator, if it is read. If end-of-file is encountered as the first character, EOF is returned, otherwise str is returned. The *fputs( )* function returns a non-negative number or EOF if unsuccessful.

Example 12.2

Write a program read a file and count the number of lines in the file, assuming that a line can contain at most 80 characters.

```
/*Program to read a file and count the number of lines in the file */
#include<stdio.h>
#include<conio.h>
#include<process.h>
void main()
{
  FILE *fp;
  int cnt=0;
  char str[80];
/* open a file in read mode */

  if ((fp=fopen("lines.dat","r"))== NULL)
  {    printf("File does not exist\n");
       exit(0);
  }
/* read the file till end of file is encountered */
  while(!(feof(fp)))
  {  fgets(str,80,fp);      /*reads at most 80 characters in str */
     cnt++;                 /* increment the counter after reading a line */
  }
}/* print the number of lines */
printf("The number of lines in the file is :%d\n",cnt);
fclose(fp);
}
```

OUTPUT

Let us assume that the contents of the file "***lines.dat***" are as follows:

```
This is C programming.
I love C programming.

To be a good programmer one should have a good logic. This is a must.
C is a procedural programming language.
```

After the execution the output would be:

```
The number of lines in the file is: 4
```

12.3.3 Formatted Input/Output Functions

If the file contains data in the form of digits, real numbers, characters and strings, then character input/output functions are not enough as the values would be read in the form of characters. Also if we want to write data in some specific format to a file, then it is not possible with the above described functions. Hence C provides a set of formatted input/output functions. These are defined in standard library and are discussed below:

fscanf() and ***fprintf()***

These functions are used for formatted input and output. These are identical to *scanf()* and *printf()* except that the first argument is a file pointer that specifies the file to be read or written, the second argument is the format string. The syntax for these functions is:

*int fscanf(FILE *fp, char *format,. . .);*
*int fprintf(FILE *fp, char *format,. . .);*

Both these functions return an integer indicating the number of bytes actually read or written.

Example 12.3

Write a program to read formatted data (account number, name and balance) from a file and print the information of clients with zero balance, in formatted manner on the screen.

```
/* Program to read formatted data from a file */

#include<stdio.h>
main()
{
   int account;
   char name[30];
   double bal;
   FILE *fp;

 if((fp=fopen("bank.dat","r"))== NULL)
          printf("FILE not present \n");
      else
          do{
            fscanf(fp,"%d%s%lf",&account,name,&bal);
             if(!feof(fp))
                 {
                 if(bal==0)
                 printf("%d %s %lf\n",account,name,bal);
                   }
             }while(!feof(fp));
 }
```

OUTPUT

This program opens a file "***bank.dat***" in the read mode if it exists, reads the records and prints the information (account number, name and balance) of the zero balance records.

Let the file be as follows:

```
101      nuj       1200
102      Raman  1500
103      Swathi  0
104      Ajay     1600
105      Udit     0
```

The output would be as follows:

```
103      Swathi  0
105      Udit     0
```

12.3.4 Block Input/Output Functions

Block Input / Output functions read/write a block (specific number of bytes from/to a file. A block can be a record, a set of records or an array. These functions are also defined in standard library and are described below.

- *fread( )*
- *fwrite( )*

These two functions allow reading and writing of blocks of data. Their syntax is:

*int fread(void *buf, int num_bytes, int count, FILE *fp);*

*int fwrite(void *buf, int num_bytes, int count, FILE *fp);*

In case of *fread(),* buf is the pointer to a memory area that receives the data from the file and in *fwrite(),* it is the pointer to the information to be written to the file. *num_bytes* specifies the number of bytes to be read or written. These functions are quite helpful in case of binary files. Generally these functions are used to read or write array of records from or to a file. The use of the above functions is shown in the following program.

Example 12.4

Write a program using *fread( )* and *fwrite()* to create a file of records and then read and print the same file.

```
/* Program to illustrate the fread() and fwrite() functions*/
#include<stdio.h>
#include<conio.h>
#include<process.h>
#include<string.h>

void main()
{
  struct stud
          {
           char name[30];
           int age;
           int roll_no;
          }s[30],st;
  int i;
  FILE *fp;

/*opening the file in write mode*/
         if((fp=fopen("sud.dat","w"))== NULL)
           {   printf("Error while creating a file\n");
                       exit(0);    }

/* reading an array of students */
           for(i=0;i<30;i++)
          scanf("%s %d %d",s[i].name,s[i].age,s[i].roll_no);

 /* writing to a file*/
          fwrite(s,sizeof(struct stud),30,fp);
          fclose(fp);

/* opening a file in read mode */
          fp=fopen("stud.dat","r");

/* reading from a file and writing on the screen */
          while(!feof(fp))
          {
            fread(&st,sizeof(struct stud),1,fp);
            fprintf("%s %d %d",st.name,st.age,st.roll_no);
           }
          fclose(fp);        }
```

OUTPUT

This program reads 30 records (name, age and roll_number) from the user, writes one record at a time to a file. The file is closed and then reopened in read mode; the records are again read from the file and written on to the screen.

Check Your Progress 2

1. Give the output of the following code fragment:

```
#include<stdio.h>
#include<process.h>
#include<conio.h>
main()
{
FILE * fp1, * fp2;
        double a,b,c;

fp1=fopen("file1", "w");
fp2=fopen("file2", "w");

fprintf(fp1,"1 5.34 –4E02");
fprintf(fp2,"-2\n1.245\n3.234e02\n");
     fclose(fp1);
     fclose(fp2);

fp1=fopen("file1", "r");
fp2=fopen("file2","r");

 fscanf(fp1,"%lf %lf %lf",&a,&b,&c);
 printf("%10lf %10lf %10lf",a,b,c);
 fscanf(fp2,"%lf %lf %lf",&a,&b,&c);
 printf("%10.1e %10lf %10lf",a,b,c);

      fclose(fp1);
      fclose(fp2);
 }
```

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

2. What is the advantage of using fread/fwrite functions?

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

3. ________ and _______ functions are used for formatted input and output from a file.

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

12.4 SEQUENTIAL Vs RANDOM ACCESS FILES

We have seen in section 12.0 that C supports two type of files – text and binary files, also two types of file systems – buffered and unbuffered file system. We can also differentiate in terms of the type of file access as Sequential access files and random access files. Sequential access files allow reading the data from the file in sequential manner which means that data can only be read in sequence. All the above examples that we have considered till now in this unit are performing sequential access. Random access files allow reading data from any location in the file. To achieve this purpose, C defines a set of functions to manipulate the position of the file pointer. We will discuss these functions in the following sections.

12.5 POSITIONING THE FILE POINTER

To support random access files, C requires a function with the help of which the file pointer can be positioned at any random location in the file. Such a function defined in the standard library is discussed below:

The function *fseek( )* is used to set the file position. Its prototype is:

*int fseek(FILE *fp, long offset, int pos);*

The first argument is the pointer to a file. The second argument is the number of bytes to move the file pointer, counting from zero. This argument can be positive, negative or zero depending on the desired movement. The third parameter is a flag indicating from where in the file to compute the offset. It can have three values:

SEEK_SET(or value 0)	the beginning of the file,
SEEK_CUR(or value 1)	the current position and
SEEK_END(or value 2)	the end of the file

These three constants are defined in <*stdio.h*>. If successful *fseek( )* returns zero. Another function *rewind()* is used to reset the file position to the beginning of the file. Its prototype is:

*void rewind(FILE *fp);*

A call to rewind is equivalent to the call

fseek(fp,0,SEEK_SET);

Another function *ftell()* is used to tell the position of the file pointer. Its prototype is:

*long ftell(FILE *fp);*

It returns –1 on error and the position of the file pointer if successful.

Example 12.5

Write a program to search a record in an already created file and update it. Use the same file as created in the previous example.

```
/*Program to search a record in an already created file*/

#include<stdio.h>
#include<conio.h>
#include<stdio.h>
#include<process.h>
#include<string.h>
void main()
{
        int r,found;
        struct stud
         {
            char name[30];
            int age;
            int roll_no;
           }st;
         FILE *fp;
        /* open the file in read/write mode */

          if((fp=fopen("f1.dat","r+b"))==NULL)
         {  printf("Error while opening the file \n");
            exit(0);
          }

/* Get the roll_no of the student */
        printf("Enter the roll_no of the record to be updated\n");
        found=0;
        scanf("%d",&r);

 /* check in the file for the existence of the roll_no */
        while((!feof(fp)) && !(found))
        {   fread(&st,sizeof(stud),1,fp);
            if(st.roll_no == r)

 /* if roll_no is found then move one record   backward to update it */
            {   fseek(fp,- sizeof(stud),SEEK_CUR);
                printf("Enter the new name\n");
                scanf("%s",st.name);
                fwrite(fp,sizeof(stud),1,fp);
                found=1;
             }
         }
         if (!found)
            printf("Record not present\n");
         fclose(fp);
        }
```

OUTPUT

Let the input file be as follows:

```
Geeta    18       101
Leena    17       102
Mahesh 23         103
Lokesh  21        104
Amit     19       105
```

Let the roll_no of the record to be updated be 106. Now since this roll_no is not present the output would be:

```
Record not present
```

If the roll_no to be searched is 103, then if the new name is Sham, the output would be the file with the contents:

```
Geeta    18       101
Leena    17       102
Sham     23       103
Lokesh  21        104
Amit     19        105
```

12.6 THE UNBUFFERED I/O – THE UNIX LIKE FILE ROUTINES

The buffered I/O system uses buffered input and output, that is, the operating system handles the details of data retrieval and storage, the system stores data temporarily (buffers it) in order to optimize file system access. The buffered I/O functions are handled directly as system calls without buffering by the operating system. That is why they are also known as low level functions. This is referred to as unbuffered I/O system because the programmer must provide and maintain all disk buffers, the routines do not do it automatically.

The low level functions are defined in the header file <*io.h*>.

These functions do not use file pointer of type FILE to access a particular file, but they use directly the file descriptors**,** as explained earlier, of type integer**.** They are also called *handles***.**

Opening and closing of files

The function used to open a file is *open( ).* Its prototype is:

int open(char *filename, int mode, int access);

Here *mode* indicates one of the following macros defined in <*fcntl.h*>.

Mode:

O_RDONLY	Read only
O_WRONLY	Write only
O_RDWR	Read / Write

The *access* parameter is used in UNIX environment for providing the access to particular users and is just included here for compatibility and can be set to zero. o*pen()* function returns *–1* on failure. It is used as:

Code fragment 2

```
int fd;

if ((fd=open(filename,mode,0)) == -1)
 {    printf("cannot open file\n");
      exit(1);   }
```

If the file does not exist, *open()* the function will not create it. For this, the function *creat()* is used which will create new files and re-write old ones. The prototype is:

*int creat(char *filename, int access);*

It returns a file descriptor; if successful else it returns –1. It is not an error to create an already existing file, the function will just truncate its length to zero. The *access* parameter is used to provide permissions to the users in the UNIX environment.
The function *close()* is used to close a file. The prototype is:

int close(int fd);

It returns zero if successful and –1 if not.

Reading, Writing and Positioning in File

The functions *read()* and *write()* are used to read from and write to a file. Their prototypes are:

*int read(int fd, void *buf, int size);*
*int write(int fd, void *buf, int size);*

The first parameter is the file descriptor returned by *open()*, the second parameter holds the data which must be typecast to the format needed by the program, the third parameter indicates the number of bytes to transferred. The return value tells how many bytes are actually transferred. If this value is –1, then an error must have occurred.

Example 12.6

Write a program to copy one file to another to illustrate the use of the above functions. The program should expect two command line arguments indicating the name of the file to be copied and the name of the file to be created.

```
/* Program to copy one file to another file to illustrate the functions*/
# include<stdio.h>
# include<io.h>
#include<process.h>

typedef char arr[80];
typedef char name[30];

main()
{
arr buf;
name fname, sname;
int fd1,fd2,size;

 /* check for the command line arguments */
if (argc!=3)
 {   printf("Invalid number of arguments\n");
          exit(0);
 }
  if ((fd1=open(argv[1],O_RDONLY))<0)
     {   printf("Error in opening file %s \n",argv[1]);
          exit(0);
         }
if ((fd2=creat(argv[2],0))<0)
        {   printf("Error in creating file %s \n",argv[2]);
          exit(0);}

 open(argv[2],O_WRONLY);
 size=read(fd1,buf,80);  /* read till end of file */

while (size>0)
       {  write(fd2,buf,80);
         size=read(fd1,buf,80);
       }
       close(fd1);
       close(fd2);
}
```

OUTPUT

If the number of arguments given on the command line is not correct then output would be:

Invalid number of arguments

One file is opened in the read mode, and another file is opened in the write mode. The output would be as follows is the file to be read is not present (let the file be *f1.dat*):

Error in opening file *f1.dat*

The output would be as follows if the disk is full and the file cannot be created (let the output file be *f2.dat*):

Error in creating file *f2.dat*

If there is no error contents of *f1.dat* will be copied to *f2.dat*.

lseek()

The function *lseek()* is provided to move to the specific position in a file. Its prototype is:

long lseek(int fd, long offset, int pos);

This function is exactly the same as *fseek()* except that the file descriptor is used instead of the file pointer.

Using the above defined functions, it is possible to write any kind of program dealing with files.

Check Your Progress 3

1. Random access is possible in C files using function ____________.

2. Write a proper C statement with proper arguments that would be called to move the file pointer back by 2 bytes.

…………………………………………………………………………………………

…………………………………………………………………………………………

3. Indicate the header files needed to use unbuffered I/O.

…………………………………………………………………………………………

…………………………………………………………………………………………

……………………………………………………………………………………..….

12.7 SUMMARY

In this unit, we have learnt about files and how C handles them. We have discussed the buffered as well as unbuffered file systems. The available functions in the standard library have been discussed. This unit provided you an ample set of programs to start with. We have also tried to differentiate between sequential access as well as random access file. The file pointers assigned to standard input, standard output and standard error are *stdin, stdout,* and *stderr* respectively. The unit clearly explains the different type of modes oof opening the file. As seen there are several functions available to read/write from the file. The usage of a particular function depends on the application. After reading this unit one must be able to handle large data bases in the form of files.

12.8 SOLUTIONS / ANSWERS

Check Your Progress 1

1. *fopen()* function links a file to a stream by returning a pointer to a FILE structure defined in <*stdio.h*>. This structure contains an index called file descriptor to a File Control Block, which is maintained by the operating system for administrative purposes.

2. Text files and binary files differ in terms of storage. In text files everything is stored in terms of text while binary files stores exact memory image of the data i.e. in text files 154 would take 3 bytes of storage while in binary files it will take 2 bytes as required by an integer.

3. *EOF* is an end-of-file marker. It is a macro defined in <*stdio.h*>. Its value is –1.

Check Your progress 2

1. The output would be:

   ```
   1.000000 5.340000 –400.000000 -2.0e+00 1.245000 323.400000
   ```

2. The advantage of using these functions is that they are used for block read/write, which means we can read or write a large set of data at one time thus increasing the speed.

3. *fscanf()* and *fprintf()* functions are used for formatted input and output from a file.

Check Your progress 3

1. Random access is possible in C files using function *fseek()*.

2. fseek(fp, -2L, SEEK_END);

3. <io.h> and <fcntl.h>

12.9 FURTHER READINGS

1. The C Programming Language, *Kernighan & Richie*, PHI Publication, 2002.
2. C How to Program, *Deitel & Deitel*, Pearson Education, 2002.
3. Practical C Programming, *Steve Oualline*, Oreilly Publication, 2003.

APPENDIX-A

THE ASCII SET

The ASCII (American Standard Code for Information Interchange) character set defines 128 characters (0 to 127 decimal, 0 to FF hexadecimal, and 0 to 177 octal). This character set is a subset of many other character sets with 256 characters, including the ANSI character set of MS Windows, the Roman-8 character set of HP systems, and the IBM PC Extended Character Set of DOS, and the ISO Latin-1 character set used by Web browsers. They are not the same as the EBCDIC character set used on IBM mainframes. The first 32 values are non-printing **control characters**, such as *Return* and *Line feed*. You generate these characters on the keyboard by holding down the Control key while you strike another key. For example, Bell is value 7, Control plus G, often shown in documents as ^G. Notice that 7 is 64 less than the value of G (71); the Control key subtracts 64 from the value of the keys that it modifies. The table shown below gives the list of the control and printing characters.

The Control Characters

Char	Oct	Dec	Hex	Control-Key	Control Action
NUL	0	0	0	^@	Null character
SOH	1	1	1	^A	Start of heading, = console interrupt
STX	2	2	2	^B	Start of text, maintenance mode on HP console
ETX	3	3	3	^C	End of text
EOT	4	4	4	^D	End of transmission, not the same as ETB
ENQ	5	5	5	^E	Enquiry, goes with ACK; old HP flow control
ACK	6	6	6	^F	Acknowledge, clears ENQ logon hand
BEL	7	7	7	^G	Bell, rings the bell...
BS	10	8	8	^H	Backspace, works on HP terminals/computers
HT	11	9	9	^I	Horizontal tab, move to next tab stop
LF	12	10	a	^J	Line Feed
VT	13	11	b	^K	Vertical tab
FF	14	12	c	^L	Form Feed, page eject
CR	15	13	d	^M	Carriage Return
SO	16	14	e	^N	Shift Out, alternate character set
SI	17	15	f	^O	Shift In, resume defaultn character set
DLE	20	16	10	^P	Data link escape
DC1	21	17	11	^Q	XON, with XOFF to pause listings; ":okay to send".
DC2	22	18	12	^R	Device control 2, block-mode flow control
DC3	23	19	13	^S	XOFF, with XON is TERM=18 flow control
DC4	24	20	14	^T	Device control 4
NAK	25	21	15	^U	Negative acknowledge
SYN	26	22	16	^V	Synchronous idle
ETB	27	23	17	^W	End transmission block, not the same as EOT
CAN	30	24	17	^X	Cancel line, MPE echoes !!!
EM	31	25	19	^Y	End of medium, Control-Y interrupt
SUB	32	26	1a	^Z	Substitute
ESC	33	27	1b	^[	Escape, next character is not echoed
FS	34	28	1c	^\	File separator
GS	35	29	1d	^]	Group separator
RS	36	30	1e	^^	Record separator, block-mode terminator
US	37	31	1f	^_	Unit separator

Printing Characters

Char	Octal	Dec	Hex	Description
SP	40	32	20	Space
!	41	33	21	Exclamation mark
"	42	34	22	Quotation mark (&quot; in HTML)
#	43	35	23	Cross hatch (number sign)
$	44	36	24	Dollar sign
%	45	37	25	Percent sign
&	46	38	26	Ampersand
`	47	39	27	Closing single quote (apostrophe)
(	50	40	28	Opening parentheses
)	51	41	29	Closing parentheses
*	52	42	2a	Asterisk (star, multiply)
+	53	43	2b	Plus
,	54	44	2c	Comma
-	55	45	2d	Hyphen, dash, minus
.	56	46	2e	Period
/	57	47	2f	Slant (forward slash, divide)
0	60	48	30	Zero
1	61	49	31	One
2	62	50	32	Two
3	63	51	33	Three
4	64	52	34	Four
5	65	53	35	Five
6	66	54	36	Six
7	67	55	37	Seven
8	70	56	38	Eight
9	71	57	39	Nine
:	72	58	3a	Colon
;	73	59	3b	Semicolon
<	74	60	3c	Less than sign (&lt; in HTML)
=	75	61	3d	Equals sign
>	76	62	3e	Greater than sign (&gt; in HTML)
?	77	63	3f	Question mark
@	100	64	40	At-sign
A	101	65	41	Uppercase A
B	102	66	42	Uppercase B
C	103	67	43	Uppercase C
D	104	68	44	Uppercase D
E	105	69	45	Uppercase E
F	106	70	46	Uppercase F
G	107	71	47	Uppercase G
H	110	72	48	Uppercase H
I	111	73	49	Uppercase I
J	112	74	4a	Uppercase J
K	113	75	4b	Uppercase K
L	114	76	4c	Uppercase L
M	115	77	4d	Uppercase M
N	116	78	4e	Uppercase N

O	117	79	4f	Uppercase O
P	120	80	50	Uppercase P
Q	121	81	51	Uppercase Q
R	122	82	52	Uppercase R
S	123	83	53	Uppercase S
T	124	84	54	Uppercase T
U	125	85	55	Uppercase U
V	126	86	56	Uppercase V
W	127	87	57	Uppercase W
X	130	88	58	Uppercase X
Y	131	89	59	Uppercase Y
Z	132	90	5a	Uppercase Z
[	133	91	5b	Opening square bracket
\	134	92	5c	Reverse slant (Backslash)
]	135	93	5d	Closing square bracket
^	136	94	5e	Caret (Circumflex)
_	137	95	5f	Underscore
`	140	96	60	Opening single quote
a	141	97	61	Lowercase a
b	142	98	62	Lowercase b
c	143	99	63	Lowercase c
d	144	100	64	Lowercase d
e	145	101	65	Lowercase e
f	146	102	66	Lowercase f
g	147	103	67	Lowercase g
h	150	104	68	Lowercase h
i	151	105	69	Lowercase i
j	152	106	6a	Lowercase j
k	153	107	6b	Lowercase k
l	154	108	6c	Lowercase l
m	155	109	6d	Lowercase m
n	156	110	6e	Lowercase n
o	157	111	6f	Lowercase o
p	160	112	70	Lowercase p
q	161	113	71	Lowercase q
r	162	114	72	Lowercase r
s	163	115	73	Lowercase s
t	164	116	74	Lowercase t
u	165	117	75	Lowercase u
v	166	118	76	Lowercase v
w	167	119	77	Lowercase w
x	170	120	78	Lowercase x
y	171	121	79	Lowercase y
z	172	122	7a	Lowercase z
{	173	123	7b	Opening curly brace
|	174	124	7c	Vertical line
}	175	125	7d	Cloing curly brace
~	176	126	7e	Tilde (approximate)
DEL	177	127	7f	Delete (rubout), cross-hatch box