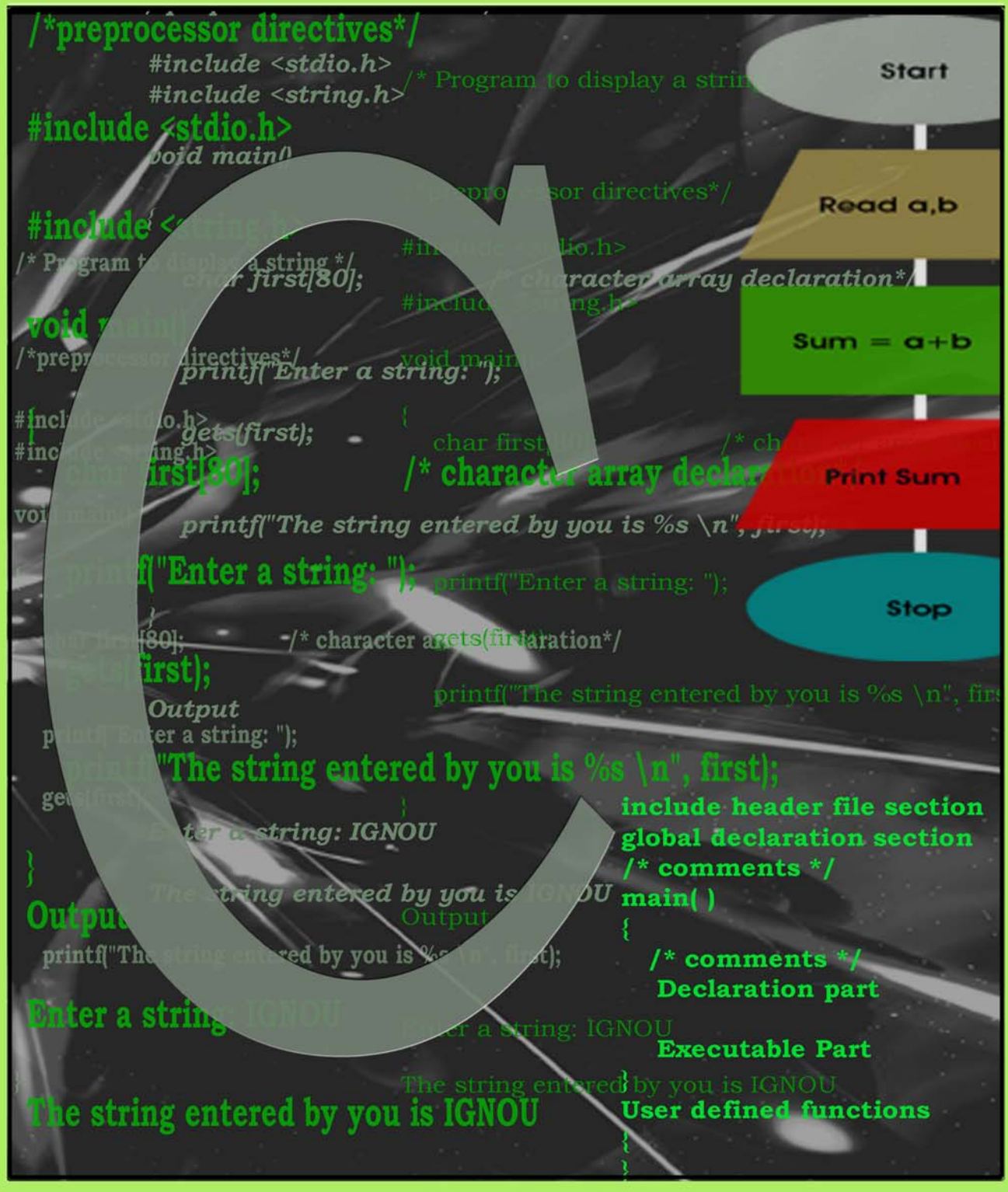
Indira Gandhi National Open University
School of Computer and Information Sciences

MCS - 011
PROBLEM SOLVING AND PROGRAMMING



An Introduction to C

1

Indira Gandhi
National Open University
School of Computer and
Information Sciences

MCS-011
PROBLEM SOLVING AND PROGRAMMING

Block

1

AN INTRODUCTION TO C

Programme / Course Design Committee

Prof. Sanjeev K. Aggarwal, IIT, Kanpur
Prof. M. Balakrishnan, IIT , Delhi
Prof Harish Karnick, IIT, Kanpur
Prof. C. Pandurangan, IIT, Madras
Dr. Om Vikas, Sr. Director, MIT
Prof P. S. Grover, Sr. Consultant,
SOCIS, IGNOU

Faculty of School of Computer and Information Sciences
Shri Shashi Bhushan
Shri Akshay Kumar
Prof Manohar Lal
Shri V.V. Subrahmanyam
Shri P.Venkata Suresh

Block Preparation Team

Prof P. S. Grover (Content Editor)
(Sr Consultant
SOCIS, IGNOU)

Ms. Charu Devgon
Dept. of Computer Science
A N Dev College
University of Delhi

Ms. Namita Gupta
Dept. of Computer Science
Maharaja Agrasen Institute
of Technology
Delhi

Ms. Priti Sehgal
Dept. of Computer Science
Keshav Mahavidyalya
University of Delhi

Shri V.V. Subrahmanyam
SOCIS, IGNOU

Shri S.S. Rana
New Delhi

Prof Sunaina Kumar,
SOH, IGNOU

Language Editors

Course Coordinator : V.V. Subrahmanyam

Block Production Team

Shri H.K Som, SOCIS

Acknowledgements

To all the faculty of SOCIS, IGNOU for their comments on the course material; to Shri Vikas Kumar for help in finalizing the CRC.

April, 2004

ISBN-81-266-1201-0

Printed and published on behalf on the Indira Gandhi National Open University, New Delhi by Director, SOCIS

COURSE INTRODUCTION

"The best way to escape from a problem is to solve it."
- Alan Saporta

Solving a problem means finding a way out of a difficulty, a way around an obstacle, attaining an aim which was not immediately attainable. Solving problems is the specific achievement of intelligence, and intelligence is the specific gift of mankind: solving problems can be regarded as the most characteristically human activity… It is a practical art, like swimming, or skiing, or playing the piano: you learn it only by imitation and practice. ... if you wish to learn swimming you have to go into the water, and if you wish to become a problem solver you have to solve problems.

Today, the usage of computers appears everywhere, especially in the educational, business, research and especially in scientific fields. Computer usage is a powerful resource and one that will grow and eventually become an integral part of all our lives. A computer is simply a logic machine that performs computations and calculations at an incredible speed. Introducing the usage of computers to solve problems, basic programming skills, and encouraging students to think logically about solving problems will help them immensely when they enter the job market. The purpose of this course is to act as an introduction to the thinking world of computers, to help students develop the logic, ability to solve the problems efficiently.

Students in every discipline need to appreciate computers not only for the applications they can run, but also for the problems they can solve. Problem Solving with Computers introduces the concepts of algorithms, iteration and conditionals, top-down design, testing and debugging and analysis of solution using any programming language or any package.

Knowledge in a programming language is prerequisite to the study of most of computer science courses. This knowledge area consists of those skills and concepts that are essential to problem solving and programming practice independent of the underlying paradigm. All programming languages will share one similarity, i.e., all are based on logic. A programming language is a programmer's principal interface with the computer. More than just knowing how to program in a single language, programmers need to understand the different styles of programming promoted by different languages. Later in your professional life, you will be working with many different programming languages and styles at once, and will encounter many different languages over the course of your career. Understanding the variety of programming languages and the design tradeoffs between the different programming paradigms makes it much easier to master new languages quickly.

We will discuss various concepts about learning and techniques for problem solving; emphasizing structured programming. In this course, you will learn concepts of computer programming, and how to implement those ideas using C programs.

This course introduces you the fundamental techniques of C programming as a foundation for more advanced study of computer science. We will discuss various concepts about learning and techniques for problem solving, emphasizing structured programming. Our goal will be to improve your reasoning and thinking skills, which should prove helpful, not only in future programming, but throughout your academic and professional career. In this course, the topics include the problem solving strategies, concept of an algorithm, flowcharting, standard C

programming constructs, primitive data types and fundamental data structures (strings, arrays, and structures and unions), files and pointers.

Upon completing this course, you should be able to:

- apply problem solving techniques;
- design algorithms and flowcharts;
- develop a working knowledge of the C language and its uses, characteristics, and capabilities;
- define problems and create solutions that can be converted into modular, structured C programs;
- use libraries of C functions;
- to use arrays and structures to organize data;
- make a righteous stab at understanding pointers;
- read and write data to and from files.

This course consists of 3 blocks and is organized in the following manner:

Block - 1 covers the problem solving skills and the basic programming constructs of C .

Block - 2 covers the control statements, arrays, strings and functions

Block - 3 covers the structures, pointers and the file handling in C.

After going through this course, you should be able to write fairly complex programs using C programming language. To get the maximum benefit from this, it is necessary that you should understand and execute all the example programs given in this course, as well, complete the assignment problems given in the lab manual also.

This course is very much helpful and first step for program development, which will be beneficial for the future courses like Data Structures, Design and Analysis of Algorithms and other programming languages.

Happy programming!

BLOCK INTRODUCTION

This block is on problem solving techniques and basic programming constructs of C language.

Problem-solving skills are recognized as an integral component of computer programming and in this block the primary focus of this course is to teach the basic programming constructs of C language. Emphasis is placed on developing the student's ability to apply problem-solving strategies to design algorithms and to implement these algorithms in a structured procedural programming language. This course includes a laboratory component also where in which the student gets the hands on experience. Basically one must explore possible avenues to a solution one by one until s/he comes across a right path to an optimize and efficient solution. In general, as one gains experience in solving problems, one develops one's own techniques and strategies, though they are often intangible.

This block consists of 4 units and is organized as follows:

Unit - 1 provides an overview of problem solving techniques, algorithm design, top – down design, analysis of algorithm efficiency and complexity.

Unit - 2 introduces the history of C language, salient features of C, structure of a C program, way to write, compile, link and run a C program.

Unit - 3 outlines the basic programming constructs of C language.

Unit - 4 provides the overview of the various operators and primitive data types in C.

programming constructs, primitive data types and fundamental data structures (strings, arrays, and structures and unions), files and pointers.

Upon completing this course, you should be able to:

- apply problem solving techniques;
- design algorithms and flowcharts;
- develop a working knowledge of the C language and its uses, characteristics, and capabilities;
- define problems and create solutions that can be converted into modular, structured C programs;
- use libraries of C functions;
- to use arrays and structures to organize data;
- make a righteous stab at understanding pointers;
- read and write data to and from files.

This course consists of 3 blocks and is organized in the following manner:

Block - 1 covers the problem solving skills and the basic programming constructs of C .

Block - 2 covers the control statements, arrays, strings and functions

Block - 3 covers the structures, pointers and the file handling in C.

After going through this course, you should be able to write fairly complex programs using C programming language. To get the maximum benefit from this, it is necessary that you should understand and execute all the example programs given in this course, as well, complete the assignment problems given in the lab manual also.

This course is very much helpful and first step for program development, which will be beneficial for the future courses like Data Structures, Design and Analysis of Algorithms and other programming languages.

Happy programming!

BLOCK INTRODUCTION

This block is on problem solving techniques and basic programming constructs of C language.

Problem-solving skills are recognized as an integral component of computer programming and in this block the primary focus of this course is to teach the basic programming constructs of C language. Emphasis is placed on developing the student's ability to apply problem-solving strategies to design algorithms and to implement these algorithms in a structured procedural programming language. This course includes a laboratory component also where in which the student gets the hands on experience. Basically one must explore possible avenues to a solution one by one until s/he comes across a right path to an optimize and efficient solution. In general, as one gains experience in solving problems, one develops one's own techniques and strategies, though they are often intangible.

This block consists of 4 units and is organized as follows:

Unit - 1 provides an overview of problem solving techniques, algorithm design, top – down design, analysis of algorithm efficiency and complexity.

Unit - 2 introduces the history of C language, salient features of C, structure of a C program, way to write, compile, link and run a C program.

Unit - 3 outlines the basic programming constructs of C language.

Unit - 4 provides the overview of the various operators and primitive data types in C.

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**.**

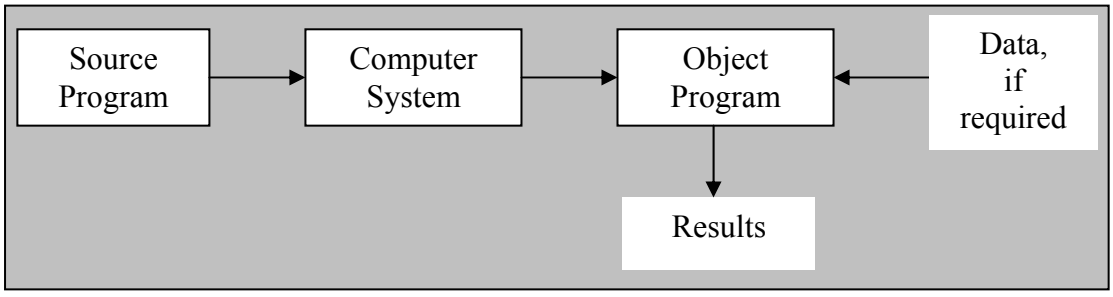
The problem-solving is a skill and there are no universal approaches one can take to solving problems. Basically one must explore possible avenues to a solution one by one until s/he comes across a right path to a solution. In general, as one gains experience in solving problems, one develops one's own 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.

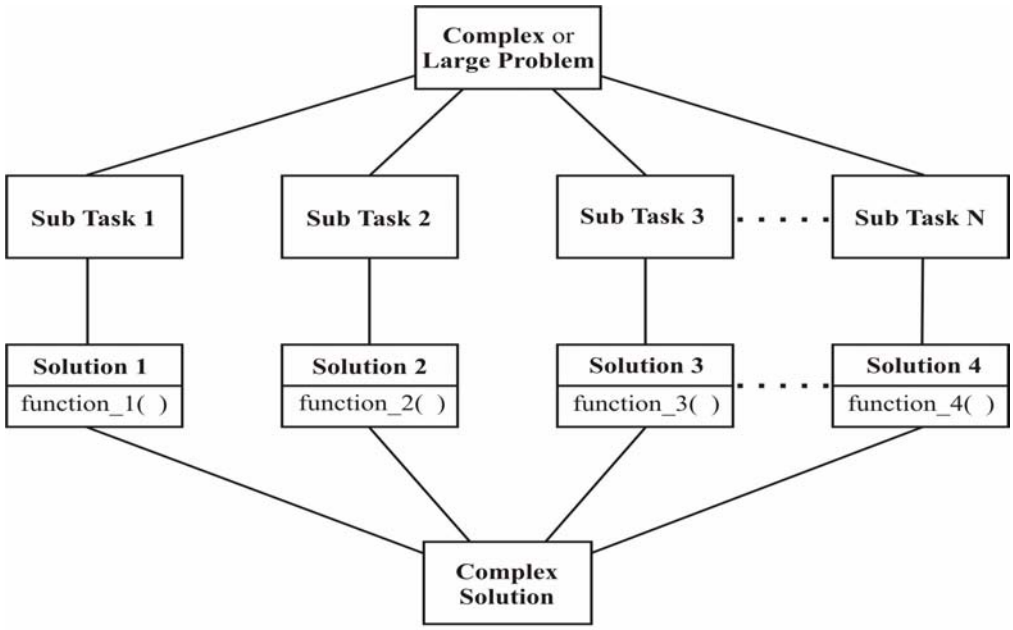


Figure 1.1: Schematic breakdown of a problem into subtasks as employed in top down design

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.

$\mathbf{Log_2\ n}$	**n**	$\mathbf{n\ log_2\ n}$	$\mathbf{n^2}$	$\mathbf{n^3}$	$\mathbf{2^n}$
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.

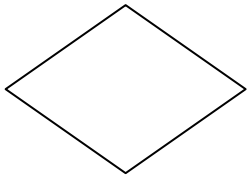
1.6 FLOWCHARTS

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

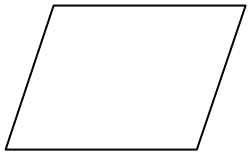
algorithm. It makes use of symbols which are connected among them to indicate the flow of information and processing. It will show the general outline of how to solve a problem or perform a task. It is prepared for better understanding of the algorithm.

1.6.1 Basic Symbols used in flowchart design

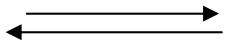
Start/Stop



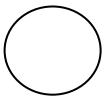
Question, Decision (Use in Branching)



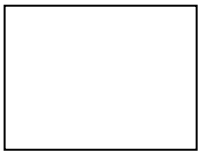
Input/Output



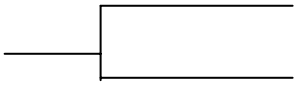
Lines or arrows represent the direction of the flow of control.



Connector (connect one part of the flowchart to another)

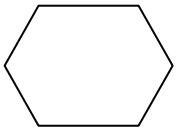


Process, Instruction

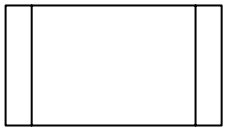


Comments, Explanations, Definitions.

Additional Symbols Related to more advanced programming



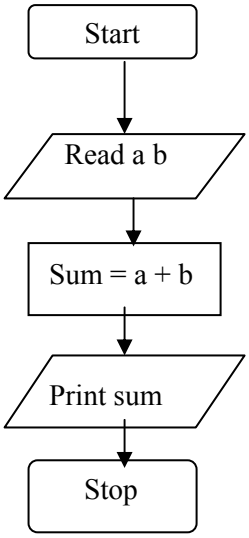
Preparation (may be used with “do Loops”)



Refers to separate flowchart

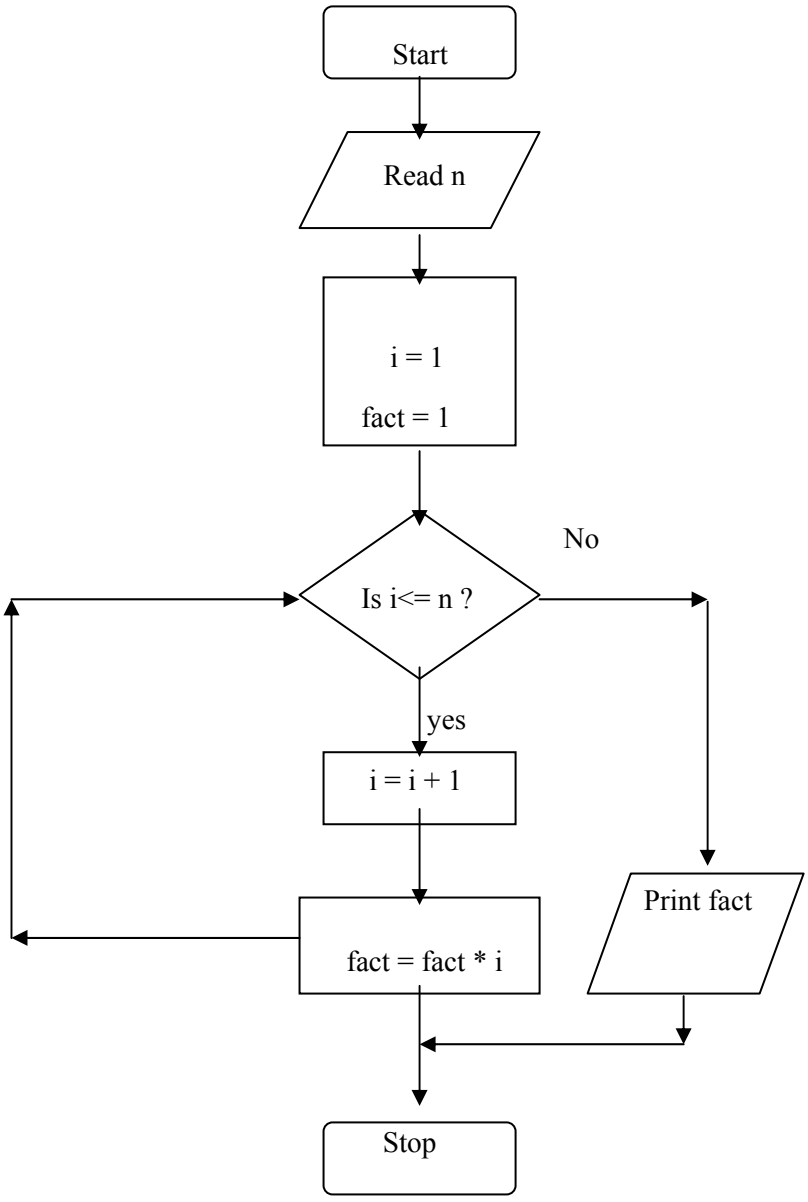
Example 1.5

The flowchart for the Example 1.1 is shown below:



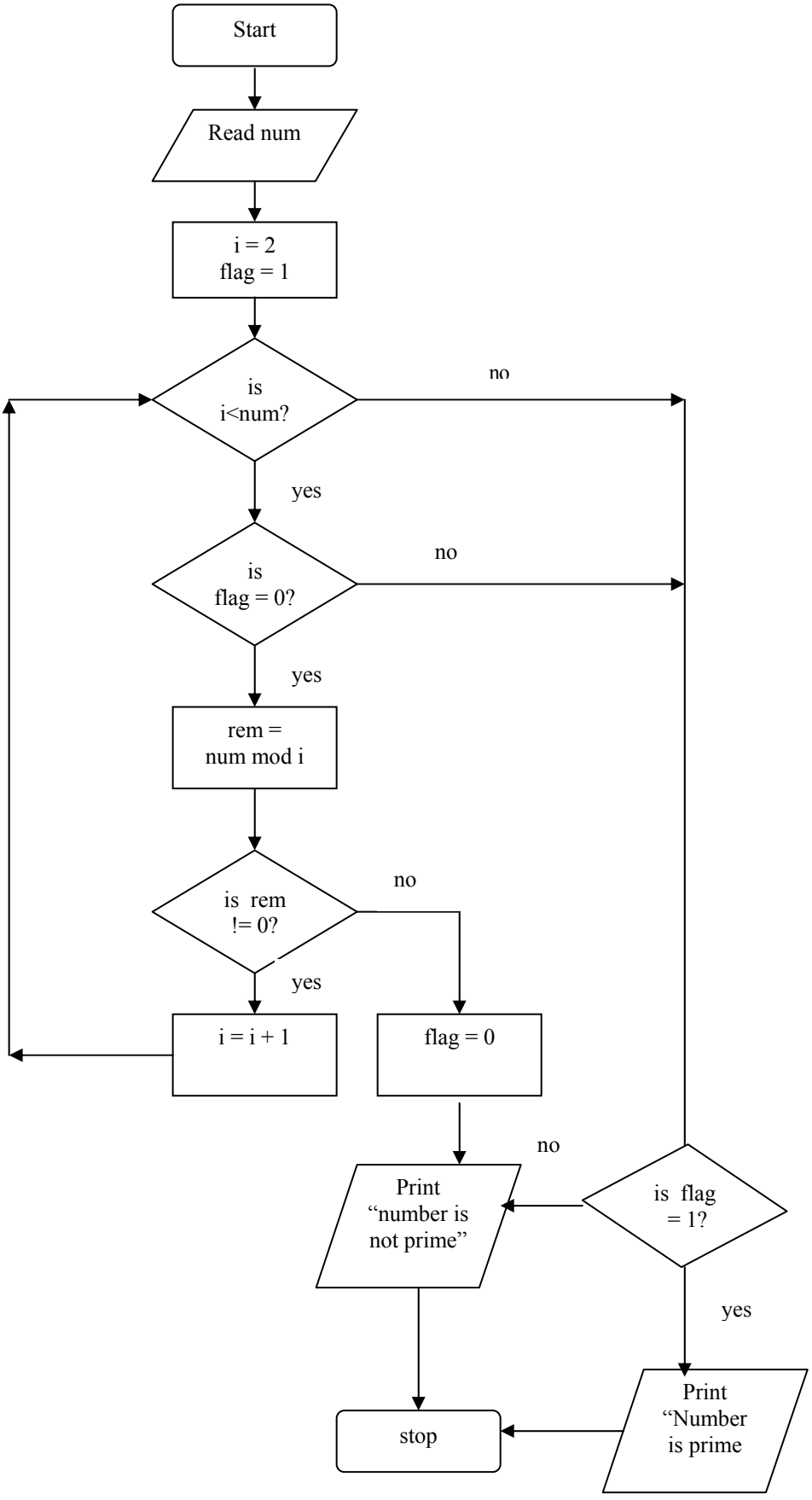
Example 1.6

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

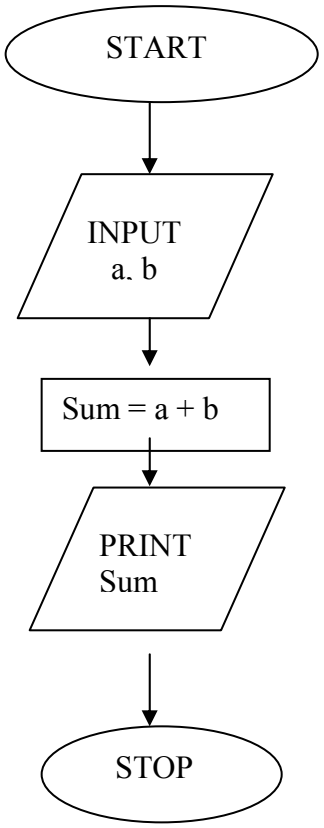


Figure 2.2: Flow chart to add two numbers

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.

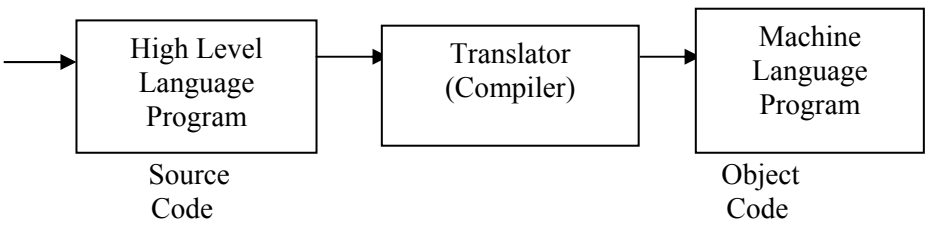


Figure 2.3: Process of Translation

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.

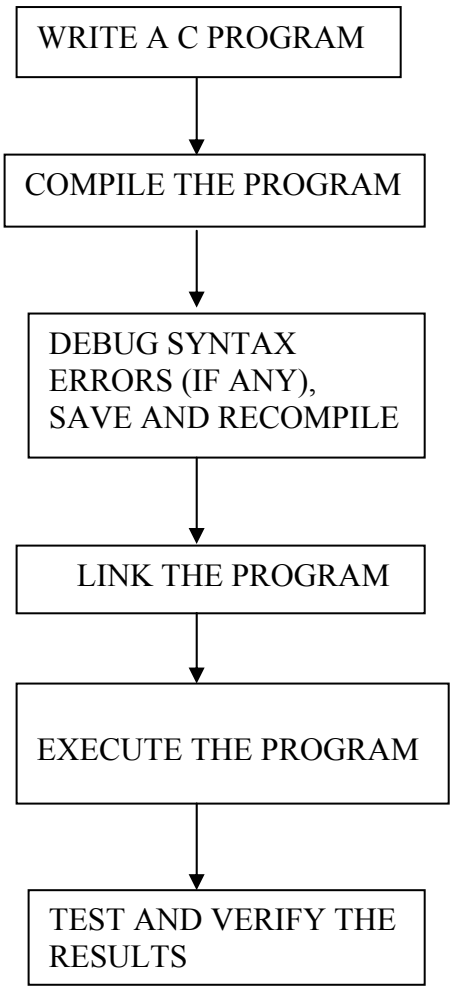


Figure 2.4: Program Execution Process

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.

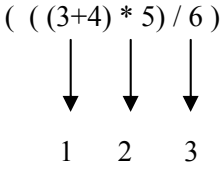
The rules of arithmetic precedence are as follows:

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.

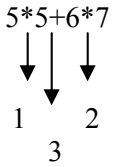


2. Multiplication, Division and Modulus operators are evaluated next. If an expression contains several multiplication, division and modulus operators, evaluation proceeds from left to right. These three are at the same level of precedence.

For example,

5*5+6*7

The order of evaluation is given below.

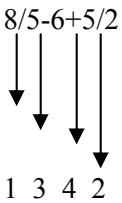


3. Addition, subtraction are evaluated last. If an expression contains several addition and subtraction operators, evaluation proceeds from left to right. Or the associativity is from left to right.

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.

SOCIAS-IGNOU/P.O.10.5T/MAY, 2004

ISBN-81-266-1230-4