‘C’ PROGRAMMING FOR MATHEMATICAL COMPUTING

STUDY MATERIAL

B.Sc. MATHEMATICS

VI SEMESTER

ELECTIVE COURSE

(2011 ADMISSION)

UNIVERSITY OF CALICUT

SCHOOL OF DISTANCE EDUCATION

THENJIPALAM, CALICUT UNIVERSITY P.O., MALAPPURAM, KERALA - 673 635

UNIVERSITY OF CALICUT

SCHOOL OF DISTANCE EDUCATION

Study Material

B.Sc.Mathematics

VI SEMESTER

ELECTIVE COURSE

C PROGRAMMING FOR MATHEMATICAL COMPUTING

Prepared by :

Dr.Valsamma K.M.
Associate Professor and Head,
Department of SAC,
Kerala Agriculture University,
KCAET, Tavanur, Malappuram.

Scrutinised by :

Dr. Anilkumar V.
Reader,
Department of Mathematics,
University of Calicut

Type settings and Lay out :
Computer Section, SDE

COURSE INTRODUCTION

The objective of this course is to introduce the basic concepts of data structure and some applications using the popular high level programming language C. Since data structure is an essential component in the development of software, the intention is to acquaint the students with a wide range of topics on this subject with appropriate example. In this course, we are discussing programming fundamentals including the programming concepts like; variables, arrays, etc., and show how all these programming concepts are used in the actual programming language called C. Being an Elective course, of this B.Sc degree (Mathematics), it is designed to complement your knowledge with C. language. The Topics of this course cover concepts on C. The course consists of Four Modules and is organized in the following manner.

Module 1:	Program Fundamentals, Algorithms and Flow charts & C Constants, variables And data types
Module 2:	Operators and Expressions and managing input / output operations.
Module 3:	Decision Making & Branching & Decision Making and looping..
Module 4:	Arrays and User defined Functions.

The first module gives a fleeting introduction to the theoretical aspects of, computer languages, operating systems, compilation and, debugging of program and to the elementary concepts like algorithms and flow charts including structure of a C program. Module **2** presents the essential features of C programming language: Variables, constants, operator types, Mathematical functions and managing of output operations. Module **3** focuses on three major decision making instructions in C, the ***if*** statement, the ***if-else*** statement and ***nested if*** and ***switch*** statement. This module also discusses the loop control instructions, the ***for***, ***while*** , ***do- while*** ,***break*** and ***continue*** statements. The last Module **4** concentrates on the sub programs i.e., functions and structured data types like arrays in detail. Concepts of arrays and user defined functions are also included.

Module I :Introduction

Welcome to the fascinating world of computer programming. This module is primarily about the implementation of computer programs using C programming language. No prior programming experience is expected from your side except some familiarity with computer hardware and software concepts. This is the first of the four modules you are going to study for the C course in your elective course for B Sc degree. The advanced concepts will be introduced in the next modules of this course. In this module, what is spotlight is only the fundamentals of C programming and all of them are presented with easy to understand explanation . The module is divided into four units.

Unit 1 introduces the elementary concepts like computer languages, computer language classification, language translators, High level languages and operating system. Apart from this, the meaning of compilation of a program, program debugging, different types of errors during program execution and program life cycle also come up for discussion.

Unit 2 presents the notion of Algorithms and flow charts, as a problem solving tool highlighting the advantages and disadvantages in brief..

Unit 3 is an overview of C, the basic structure of a C program, the programming style and the steps involved for executing a C program.

The last unit, Unit 4 enlarges on data types available in the language. Variables, declaration of variables, constants, and symbolic constants are also discussed here.

Unit 1: Program Fundamentals

Structure

1.1 Computer languages.

Language is a vehicle for communication. Spoken or natural languages are used by people the world over to express ideas / issue commands or to interact with others. Currently there are more than 6900 spoken languages in the world we live. But, there is no common natural language that connects the user and the computer system which both the user and computer can understand. Computer speaks only one language--machine code consisting of just series of 1s and 0s. A computer language is needed because a computer works only with the machine language consisting of bits and bytes. Therefore in order to interact with the machine, a user needs to study a computer language which the computer understands. Programming languages, such as C was invented because trying to write computer programs in machine code is tedious and sheer madness. Programming languages allow programs to be written in a form which is far easier to read than a series of 1s and 0s.

A programming language can be defined as any of various formal coded languages that programmers use in program writing to write instructions to a computer in a manner understood by the computer to perform the task which the programmer wants the computer to do . In this process, the most basic computer language is obviously the most Low Level language and that is the machine language that uses binary code consisting of '1' and '0' . With the help of this machine language a computer can run a program very fast without using any translator or interpreter program . On the other hand , the high-level languages such as C, or Java are much simpler and more 'English-like, but the only snag is that we need to use another program, a compiler or an interpreter, to convert the high-level code into the machine code. More generally, a language that is acceptable to a computer system is called a **programming language** and it is used by a programmer to instruct a computer what he wants the computer to do. A complete specification for a programming language includes a description of a machine or a processor for that language. Moreover, a programming language involves a computer performing some kind of computation or algorithm and possibly control external devices such as printers, disks etc. Computer languages are in general categorized into general purpose and specific purpose languages.

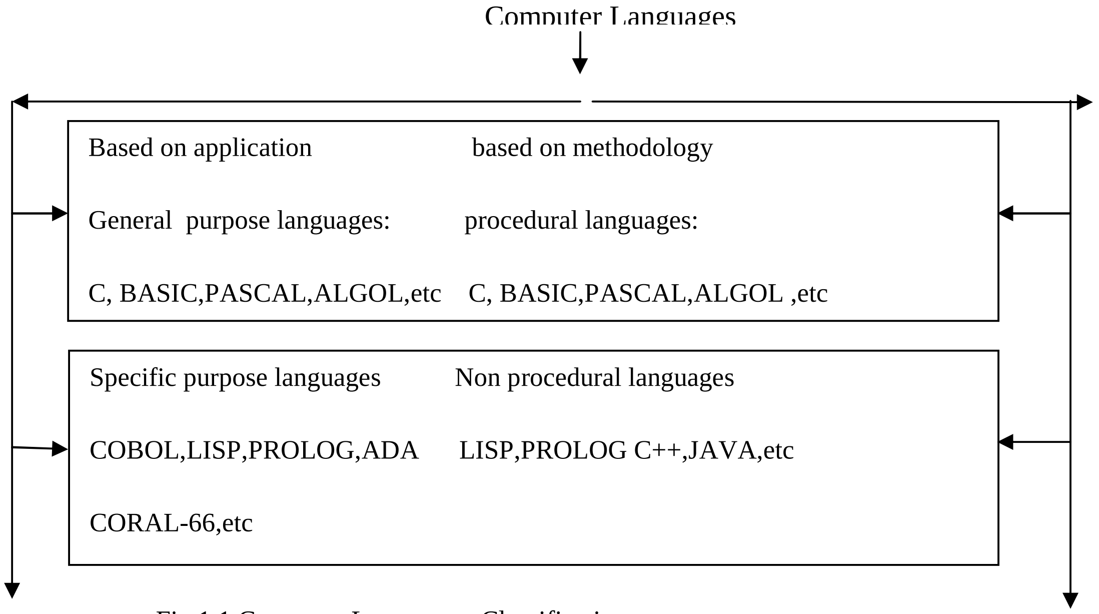


Fig 1.1 Computer Languages: Classification

1.2 Classification of Computer languages:

At present there are many computer languages, and nearly all of them have been evolving from machine language into a more natural way of writing as manifested in the high level languages. While Some languages have been adapted to the kind of application that they intended to solve, some other languages are tailor made to the specific approach used in the design. We have been using the word "generation "to indicate this evolution. High-level languages (HLL), belonging to the 3rd generation such as Pascal, FORTRAN, Algol, COBOL, PL/I, Basic, and C. are also known as Procedural languages. In procedural languages problem solving is a step by step logical process where the coded program, called a source program, has to be translated through a compilation step. But in non-procedural languages like LISP,PROLOG etc, belonging to 4th generation which are also known as 4GL the machine is instructed to obtain the results of the chosen problem without specifying how to solve it. In other words , procedural languages is concerned with, HOW and WHAT, of a process , whereas in non-procedural language it is specified what condition the answer should satisfy without specifying HOW to obtain it . In the language hierarchy, higher level languages(HLL) belonging to the 4th generation , are more English like and are much closer to human languages, where problem solving is independent of machine code of a specific computer. Now C is being increasingly used for the development of system programming applications. In general, High level languages are far simpler to understand for the humans, than the assembly level language or machine level language / language of computer (where the computer works in bits and bytes).

1.3 Language Translators in computer.

We know that a processor is a Microchip implanted in a CPU's hard drive that processes instructions sent to it by the computer and software programs. Translator is a Programming language- processor (assembler, compiler, or interpreter) that converts a computer program written in one language to another. Thus a language translator can be defined as a program or application that translates between high-level languages. It usually renders text or data format in one language into another. It is also known as a language converter or a source to source translator. The 3 language translators that translate the source programs (written in some high level language) into machine language (or machine code) are: 1) assembler 2) compiler 3) interpreter. The function of translator is brought out in fig.1.2

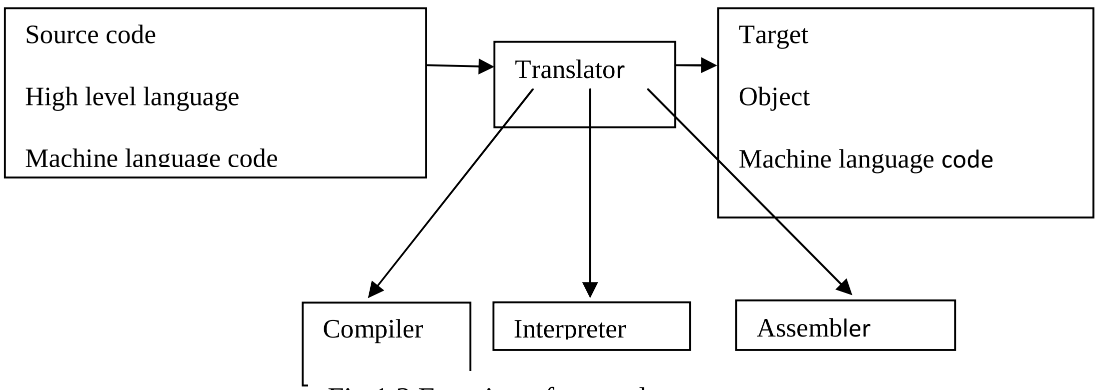


Fig 1.2 Function of a translator

Assembler is a program that takes basic computer instructions and coverts them into patterns of bits to perform the basic operations.

A Compiler is a special computer program that translates code written in a high level language to a lower level language, object / machine code by processing statements written in a particular programming language. The most common purpose behind translating a source code is to create an executable program (converting from a high level language into machine language). The task is performed by compilers by scanning the entire program first and then translating it into machine code which will be executed by the computer processor and the corresponding tasks are performed. In fact compilers make the users free from the requirements of having to know the hardware details of a computer system

An interpreter is a computer program that reads the source code of another computer program and executes that program line by line. But Line by line interpretation entails slowness in running the program and this considered a disadvantage. Each time when an interpreter gets a high level language code to be executed, it converts this code into an intermediate code before converting it into a machine code. Each part of the code is interpreted and then executed separately in a sequence and when an error is found on a part of the code, it will stop the interpreter of the code without translating the next set of codes. With all the slowness, implementing an interpreter for a language is comparatively simpler than implementing a compiler for a given language in the system. Moreover, the interpreter resides in the memory along with the program to be interpreted and controls program execution

1.4 Higher level languages .

Higher Level Languages (HLL) are advanced programming languages that enables a programmer to code programs which are more or less machine independent , in the sense the programmers do not have to worry about the intricacies of the machine architecture. HLL uses familiar English like syntax or any human language. The “Low" or "High" prefixed with the level of language indicates, how "close" to the hardware, the programming language is. A CPU normally processes either 32 or 64-bit instructions. We can visualize this as 32 ‘1’s and ‘0’s in a row that the processor interprets and executes. Writing this machine code of ‘1’s and 0’s directly into the machine would be the ’lowest-level' closest to the hardware.

In a low level language we have to care about actual memory locations, whereas in a high-level we just create variables and let the OS handle memory. In fact, HLL requires a compiler to be present on the operating system, that executes the code. The main advantage of HLL is that they are easy to work with and use and endowed with, less errors, better documentation, low program cost and save a lot of run time. Generally, HLL uses English-like statements and symbols to create sequences of computer instructions and identify memory locations, rather than using the machine-specific individual instruction codes and numerical addresses employed by machine language. But only that, HLL must be translated into its equivalent machine code before the execution of the program , using compilers. Note that , each. Computer requires separate compiler for each HLL it supports. The OS in turn request service from underlying system resources.

1.5 Operating System

Operating system(OS) is the first thing loaded on to the computer by a boot program and it is a system software that helps the user in interacting with the resources and executing the user applications. That means, it is a software program that enables the computer hardware to communicate and operate with the computer software. It also manages the other computer system resources that might be shared by different users in multiuser environments and resides in the memory just like user programs. Popular operating systems include Linux, Windows 2000, Os/400, AIX etc., and they acts like an interface between user and computer systems. The operating system uses the Basic Input Output System(BIOS) and provides a platform for the user to interact with the computer system. The BIOS is a first software run by the computer, when the system is turned On. This software is usually stored in the Read Only Memory (ROM), located on the motherboard of a computer. The main function of BIOS is to check all the hardware components attached to the computer and to load part of the OS or other system programs to RAM.

1.6 Compilation of Program

Compilation is the act of transforming the user-friendly source code written in High Level Language into the *machine readable* version of '1' and '0' that a CPU can understand and execute. A Source code must go through several steps before it can become an executable program. The first step in this direction is to run the source code through a *compiler*, that translates the high-level language instructions into machine readable object code which is actually an intermediary form of the machine language .The next step in executing the program is passing the object code through a *linker* for linking with libraries. Usually program instructions in their original form are created by the utility program called the editor which is intended to provide a paper- pencil -eraser environment on computer towards developing the source code. Since object code is not directly executable, it needs to be linked to the object file available with libraries and / or other object files or other object programs that the source code has used. The linker comes in here.

A **linker** link together a bunch of object files into a binary executable file , This includes the object files created from the source code files as well as object files that have been pre compiled and collected into library files and these files have names ending in .a. when the source code has errors the user again takes the help of the editor and corrects the errors. A syntax error free source code is then compiled into an object code. This executable code is run and the results are studied . if the results are not acceptable, the entire process is repeated .This is explained in Table 1.1 below.

.**Table : 1.1 Execution of a C Program**

STEP BY STEP EXECUTION OF C PROGRAM

STEP-1 : EDITING

- **The first step is writing C program using Text editor like Borland C,C++,Note Pad++**
- **Saving the program in .C extension**
- **Saved file in .C extension is called Source Program**

STEP-2 : COMPILING

- **Inputting C source code with [.C] extension to produce machine instruction**
- **Error free Source code gets converted into object code [.obj]**

CHECKING ERRORS

- **During step 2 compilers check for errors ,re edits & again check for errors. Error free Program gets liked with libraries.**

LINKING LIBRARIES

- **Program linked with "included header files"(** a source code file that is merged into another by preprocessor)
- Linking with other libraries executed by linker

ERROR CHECKING

- **Run time errors checked**

An **editor** is a utility program or software that appears much like a word processor, and is used to edit the source code of any program. The user makes use of an editor to type in the source code and stores it in a file suppose that the file name is test.c The character c is the default extension for the source code file. The extension *o* in UNIX and *obj* in DOS are used to indicate object code files. At the end of compilation two files are present on the disk namel test.c and test.o. This object program file is linked with system library and the other object programs to produce an executable code. The disk now contains three files test.c, test.o, a.out. By just typing a.out one can run programs

1.7 Different types of errors

Errors are very common when writing computer programs. If we want to execute a source code file, different stages like compilation linking and execution are required. A user can expect the errors indicated by the different modules running on the computer system., namely compiler linker and loader. There are the following types of Programming errors:

- Run Time Errors
- Compile Errors :syntax and semantic errors
- Logical errors

RUN TIME ERRORS:

It is an error that that occurs during the execution of a program. In programming , errors are also known as bugs and run time errors indicate bugs in the program or problems like insufficient memory or a segmentation fault caused by trying to access a memory location to that is not allowed to access Runtime errors may crash your program when you run it. Runtime errors are usually caused when a program with no syntax errors, directs the computer to perform an operation like dividing a number by zero or when the computer is instructed to find the square root of a negative integer or to find logarithm of a negative number or when there is lack of free memory space. Occurrence of these errors may stop program execution. Runtime errors are usually more difficult to find and fix than syntax errors.

COMPILE ERRORS:

Compile errors are those errors that occur at the time of compilation of the program. C compile errors may be further classified as: (i) syntax error (ii) Semantic error .

(i)Syntax errors :

Whether it is a natural language or programming language, there are a set of rules in sentence - building which govern the word order and this is called syntax. In the English language a sentence is built as per the rule of subject –verb- object agreement and violation of this rule is regarded as a syntax error. Likewise in C programming there are a set of rules for writing program statements, the violation which constitutes a syntax error. Syntax error occurs when the code is written in a manner not permitted by the rules of the program language. These errors are easily traceable at the compilation time. Error messages and flagged lines are displayed by the compiler. In Visual studio by Microsoft which supports C programming language, the error messages are made to appear in the Output window. These messages will tell the location of a syntax error (line number and file) and a short description of what the compiler thinks the error is. Some examples of syntax error are given below:-

- Missing semicolon (**;**) at the end of statement.
- Missing any of delimiters i.e **{** or **}**
- Incorrect spelling of any keyword.
- Using variable without declaration etc.

Syntax errors are the easiest to find and fix. Over the years, compiler developers have worked hard to make compilers smarter so that they can catch errors at compile time that might otherwise turn out to be runtime errors.

(ii) Semantic Errors.

The errors that occur in the logic of a program is called semantic error. It is always a violation of the rules of meaning of a natural language or programming language . When it occurs it leads to incorrect output. It can even cause the program to hang or crash. Compared with the syntax error, it is a more subtle type of error in that it cannot be easily traced. A semantic error occurs when the syntax of your code is correct, but the semantics or meaning are not what was intended. Since the syntax rules are obeyed , semantic errors are not recognized either by the compiler or interpreter, because Compilers and interpreters concern themselves only with the structure of language , not the meaning. Due to this error, the program may terminate suddenly or enter into an indefinite loop. The error diagnostic is produced by some run time systems. when such errors are fixed and corrected the correct results are produced

LOGICAL ERRORS

Logic errors are the errors in the output of the program and they occur when a programmer implements the algorithm for solving a problem incorrectly. A statement with logical error may produce unexpected and wrong results in the program. Common examples are:

- Multiplying when you should be dividing
- Adding when you should be subtracting
- Opening and using data from the wrong file

Logical errors cannot be detected by the compiler, and thus, programmers have to check the entire coding of a c program line by line.

1.8 Debugging of programs

Debugging means removing bugs from a program . A bug is an unexpected and undesirable behaviour by a program. In computers, debugging is the process of locating and fixing bugs (errors) in computer program code. The first step in removing the bug is identifying it. Some bugs are quite obvious, as when the program crashes unexpectedly. Others are obscure, as when the program produces output which is incorrect .Debugging must necessarily answer two questions : "What did the program do?" and "What did we expect the program to do?" The aim is to determine precisely the behaviour of the program to let us infer why it is not running the desirable way we wanted it to run . To *debug* a program is to isolate the source of the problem, and then fix it. Debugging is a necessary process in any new software or hardware development process, whether it is a commercial product or a personal application program. Because most computer programs and many programmed hardware devices contain thousands of lines of code, almost any new product is likely to contain a few bugs. Invariably, the bugs in the functions that get most use are found and fixed first.

The first step in fixing a bug is to replicate it. This means recreating the undesirable behaviour under controlled conditions. The aim is to find by way of precise steps, the presence of the bug. In many cases this method is straightforward. We can run the program with an input to see if the bug occurs. Debugging tools (called *debugger*s) help identify coding errors at various development stages. Some programming language packages include a facility for checking the code for errors as it is being written. Compilers can be configured to produce debug information at compile time, so that this information can be used by the debugger program to view source code as we debug. Debugging is often supported by a software tool in most of the IDE. Using such tools the user can investigate the program behavior by introducing break points in the program. At every break points the system. Suspends execution so as to facilitate the checking of intermediate results. One can even modify the values of the program variables when the break point is reached. After debugging these break point are to be removed so that the program can be subjected to normal compilation and later execution.

1.9 Rewriting and program maintenance

During debugging the user or programmer sometimes makes a large amount of refinements /alterations, and addition thus making the program appear all together like a patch work. The user will be in a fix to decide whether he may go ahead with repairing or whether it would be more useful and wiser to rewrite the program, as the total rewriting will not be as difficult as correcting the original program, since everything become clear by the time the whole program is corrected. It would be advisable to rewrite a clear version of the program and debug it rather than debugging a hopelessly repaired program. In such situations rewriting becomes more preferable than debugging .

1.10 Program maintenance stage

Updating programs or adapting to reflect changes in tune with the requirement of the new operating environments is called **Program maintenance**. In addition, it is the updating of application programs in order to meet changing information requirements, such as adding new functions and changing data formats. It also includes fixing bugs and adapting the software to new hardware devices. When a real world program is tackled, the program may change with time. The program developed should be able to absorb these changes without the compulsion of having to rewrite programs. This incorporation of changes in the program to satisfy the requirements is called program maintenance. There may be fresh errors introduced in the program at the correction stage. This activity is also covered under program maintaenance.The documentation of the program helps in its maintenance later

1.11 Program life cycle

A computer program envisages the development of a solution to an identified problem, and the setting up of a related series of instructions which, when directed through the computer hardware, will produce the desired results. First ,the problem of the end user needs to be defined so that the correct solution to solve it can be drawn up to decide what real world problem is to be solved and how a program can do this. Once a program is written, it is correct if it does what it is supposed to do. The programming development life cycle includes seven distinct stages : define, outline, develop, test, code, run, document and maintain.

This sequence must be strictly followed in order to get a good and efficient program. Problem definition originates from the user. Writing the application Program includes considerations like choosing the program language. In the first stage, the program developers must obtain the program requirements from the users and document the requirements. This may require sufficient dialogue or meetings between the end user or customer .Inputs, outputs and the processes must be known clearly. when the problem is unambiguously defined, nearly 30 % of the programming work is considered to be over.

Solution outlining stage

After understanding the requirements of the end user , the next step is to outline a strategy for solving the defined problem. This involves through system analyses and design of the entire program. This can involve looking at the requirements, flow of data, number of processes and sub processes, the best , the worst and average solutions,, etc. Amongst all the stages, this stage gives emphasis on how to try to sort out the problem on the computer system.

Algorithm and flowchart writing stage

After a programmer understands and analyzes a problem, he must come up with a solution—an algorithm. WE know that an algorithm is a step-by-step procedure for solving a problem in a finite amount of time with a finite amount of data. After outlining the solutions, the next step involves developing a detailed logic plan using tools such as algorithm and flow charts. Normally, a top down approach is used, and depending upon the need, either the algorithm for the sub program are written afresh, or the existing proven algorithms are chosen. Moreover, to help write the algorithms the programmers use other tools like flow charts which are pictorial image of the steps of an algorithm. Flow charts are a better depiction or documentation tool than an algorithm. Majority of programmers use both-the flow chart first and then the algorithm. That is, after creating the Flow chart, one writes an algorithm using the pseudo code.

Language choice stage

The choice of programming language is an important design consideration since it plays a significant role in reducing the total development time . The major factor in selecting a language is the language suitability to solve the particular classes of problems for which it is intended, and the type of the actual user (i.e. user level of professionalism).

At this stage, we translate the design into the application. That is, algorithm must be converted int o a program. Looking at the requirements of the problem, the facilities and the constraints of the language are studied. An appropriate language based on the available facilities, is chosen for implementation of the solution. The choice of language is considered as an important stage in problem solving.

Coding stage

After a program is designed, it is to be implemented. This is the stage where the algorithm is converted into a program by using selected language statements. A programming code is the program instructions written in programming language in their original form .The code that a programmer writes is called *source code*. After it has been compiled, it is called *object code*. Code that is ready to run is called *executable code* or *machine code*. Coding is one step in problem solving. Care must be taken so that the term programming should not be confused with coding. More over one should not code until the algorithm is well defined, and must be cautious in using the constructs of the language. At this stage, errors can come into play into the solution to a problem due to improper coding.

Testing stage

This is an important stage, as the acceptance of the program depends on testing and subsequent validation. As the program is being coded, and completed it must be tested out to see if it is running properly and it produces the required output with appropriate input data. The input and output must conform to the requirements That is, to make sure that the algorithm of the program does what it should be intended to do, whether the program fits as expected into the intended application environment. There are different methods of testing like ***black box, white box and Grey Box*** method of testing. The technique of testing without having any knowledge of the interior workings of the application is Black Box testing. White box testing is the detailed investigation of internal logic and structure of the code. White box testing is also called glass testing or open box testing. In order to perform white box testing on an application, the tester needs to possess knowledge of the internal working of the code. Grey Box testing is a technique to test the application with limited knowledge of the internal workings of an application. The tester needs to have a look inside the source code and find out which unit/chunk of the code is behaving inappropriately. The program will be tested for all possible input, without which it is not possible to declare that it works satisfactorily. Once testing leads to satisfactory results, the program is considered to be working correctly and is accepted.

Documentation stage

Documentation is sometimes integrated with all the stages and it is as important as the other stages. This involves the writing of small notes or memos to explain a particular portion of code , how it works, the way it performs the task, what each constant does, what is used to denote a variable, the inputs and outputs expected etc. By adding notes and memos alongside the programming, a developer can create a piece-by-piece instruction manual and description of how the application works. Documentation should also include a list of any known bugs and errors and the potential locations from where they could originate. Sufficient documentation helps others to modify and understand the program easily. This is the stage where one adds, English texts, called comments, to the program. It is always better to document the program during the development stage itself. Documentation can be either internal or external. Internal documentation is used by other programmers to help them know why you did something in a certain way or tell them how you wrote a program. External documentation on the other hand, include user manuals, FAQ's on a web, help areas, and anything that is not the actual code. That is, Original specification becomes the basis for external documentation, where as internal documentation explains how the program works.

Program Maintenance Stage:

In the Program maintenance phase programs are updated to correct to faults, improve functionality and to make changes in its execution environment. The IEEE Computer Society defines maintainability as the ease with which programs can be maintained, enhanced, adapted, or corrected to satisfy specified requirements It is the largest phase of the program life cycle .To maintain is to make sure that the program keeps running as it should. Usually, after the programs are being developed and documented, it is placed into operation. As users use the program, during their operation, either ,a program may fail to perform its objective or it must be necessary to add new functionality to a program to fix the errors or to update the program. That is, changing the program design, coding and updating are parts of the program maintenance stage. However, one can continue to fix and update the program until it reaches a point where the program has become no longer useful or too old. At that time, maintenance stops and the program development life cycle is started all over again.

1.12 Summary:

1. A **computer language** is a programming language designed for use on a specific class of computers. Programming languages are mainly of two types: High level and low level programming languages.
2. A **translato**r is a computer program, that translates a program written in a given programming language into a functionally similar program in another language without losing the essence of the program. That is, If the computer program translates a HLL in to another HLL, then it is called a **translator**. On the other hand, if the program translated assembly language to machine code then it is called an **assembler.**
3. An **operating system**, is a software that supports a computer's basic functions, such as task scheduling, execution of applications and controlling computer peripherals.
4. **Debugging** is the methodical process of identifying and removing errors or defects in a computer program by looking at lines of code one by one to see if they have been written correctly and the logic is correct.
5. **Program development life cycle**, consists of different stages such as problem definition, outlining the solution, algorithm and flow chart development, choice of language, coding testing and documentation. This sequence must be strictly followed in order to get a good and efficient program.

Unit 2: Algorithms and Flow Charts

Structure

2.1 Introduction:

To make a computer do anything, we may have to write a computer program. Again, to write a computer program, we may have to tell the computer, step by step, exactly what we want it to do. Here, we also get to decide ***how*** the computer is going to do it and with this presupposition we employ a finite step- by- step formula for problem solving. All that the computer does is "executing " the program, following each step mechanically, to accomplish the end goal .Thus an **algorithm** can also be viewed as a deterministic automaton for accomplishing a goal which, given an initial state, will terminate in a defined end-state. In programming algorithm provides the logic; data provide the values. Put together we get a Program which can be broken down as : Program = Algorithm + Data Structures . One of the principal challenges in programming is to create an elegant algorithm with fewer steps.

Without algorithm development, programming activity is considered to be incomplete. The software developer puts the body in the form of a systematic body of steps, which is then converted into a flowchart and later into a program

2.2 Algorithm and its uses

The dictionary meaning of the word **algorithm** is a process or set of rules used for calculation. In computer science the word algorithm has a historical interpretation. The word is thought to have been derived from the name of the 9th century Persian mathematician abu jafar mobammed ibn musa al-khowarzsmi. Al- khowasmi wrote a book called kitab al jabr walmukhwala which literally means “rules of restoration and reduction” . Historians of mathematics found this to be the true origin of the word algorithm

An a**lgorithm** is a precise set of rules or a precise specification of a sequence of instructions to be followed in solving problems using a computer. Each specification or instruction tells the computer what task is to be performed. One such specification is given in example below.

Example 2.1

Recipe for making Cake

Ingredients

Bengal gram flour 2 cup, sugar 2cups, ghee, 2cup, milk 1 liter, essence 6 drops, water ½ cup.

Method

Step 1 : warm up ghee. Add Bengal gram flour and fry it on slow fire for 3 minutes.

Step 2: Warm milk and add essence.

Step 3: Take ½ cup water and add 2 cups of sugar. Stir it till sugar t dissolves. Then boil it until it appears too sticky.

Step 4: Add the prepared syrup and stir it.

Step 5: Add fried Bengal gram to the syrup and stir it continuously to about 20 minutes.

Step 6: Pour the resulting mixture in a plate and allow to condense.

Step 7: After 15 minutes cut it into 20 pieces..

Result

20 pieces of cake.

In this example, the following points deserves worth noting.

1. The instructions are precise and the number of actions to be carried out are very few.
2. By taking proper permutation and combination of this set of actions we can facilitate the easy production of a cake.

The set of instruction for solving this problem is similar to algorithms , with the exception that, they do not possess all the necessary attributes of algorithms. An algorithm defined as a finite set of specifications does need the following five attributes.

1. Inputs-The inputs are to be given at the beginning, before the algorithm starts and they are to be processed by the algorithm.
2. The sequence of instructions leading to specific action in the algorithm must be precise(well ordered).
3. Each instruction must be sufficiently basic, so that it can be carried out in a finite time (effectiveness).
4. The number of repetitions to carry out a group of instruction must be finite(finiteness).
5. An algorithm must have one or more output.

Based on these five attributes , we can say that the above example does not qualify itself as an algorithm. The following is an example which illustrates how an algorithm is written to convert temperature in degree Celsius to degree Fahrenheit.

Example 2 : Converting degree Celsius to degree Fahrenheit.

Input : Temperature in degree Celsius.

Output: Temperature in degree Fahrenheit.

1. Start
2. Read temp in Celsius
3. Multiply this value by 1.8 and add 32.
4. Assign this value as degree Fahrenheit.
5. Display the result.
6. Stop.

As was discussed, this algorithm has a name , an input, an output and a body comprising instructions that begin with start and end with stop. Between this two keywords, this algorithm contains one or more steps, denoting the operations to be performed. In fact, algorithms are essential to the way computers process data. Because an algorithm is a list of precise steps, the order of computation is always critical to the functioning of the algorithm. Instructions are usually listed explicitly and are described as starting from the top- and down to the bottom. In general, an algorithm is a step by step formalization of a mapping function to map input set onto an output.

Algorithms are well-ordered

Since an algorithm is a collection of operations or instructions, we must know the correct order in which the instructions are executed. If the order is unclear, we may be uncertain which instruction should be performed next. This characteristic is especially important for computers. A computer can only execute an algorithm if it knows the exact order of steps to perform.

Algorithms have unambiguous operations

Each operation in an algorithm must be sufficiently clear so that it does not need to be simplified.. Basic operations used for writing algorithms are known as primitive operations or primitives. When an algorithm is written in computer primitives, then the algorithm is unambiguous and the computer can execute it.

Algorithms have effectively computable operations

Each operation in an algorithm must be durable, that is, the operation must be something that is possible to do. For computers, many mathematical operations such as division by zero or finding the square root of a negative number are also impossible. These operations are not effectively computable so they cannot be used in writing algorithms.

Algorithms produce a result

In our simple definition of an algorithm, we stated that an algorithm is a set of instructions for solving a problem. Unless an algorithm produces some result, we can never be certain whether our solution is correct. Only algorithms which produce results can be verified as either right or wrong.

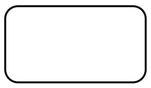
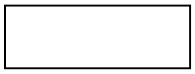
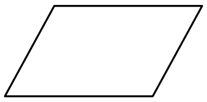
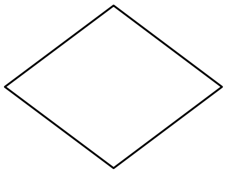
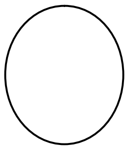
Algorithms halt in a finite amount of time

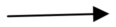
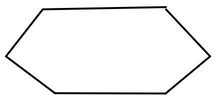
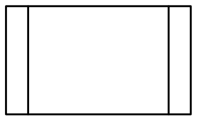
Algorithms should be composed of a finite number of operations and they should complete their execution in a finite amount of time. Every algorithm must reach some operation that tells it to stop. In the above eg, it is evident that the algorithm has a start and terminal point. Between these two key words , every algorithm contains one or more steps indicating the operations to be performed. To express algorithms one uses, structured English, pseudo programming languages and other methods .It is very convenient to express algorithm pictorially.

2.3 Flow Charts and their Uses

The flow chart is an important tool aiding development of a program.. It is usual practice to introduce another intermediate step prior to the preparation of a computer program. This step is called flow chart development. A flow chart is a pictorial or graphical representation of the steps necessary to solve a problem, perform a task, complete a process or illustrate the components of a system using certain prescribed symbols to show the sequence of operations to be performed, so as to arrive at a solution. That is, a flow chart illustrates the iterative and sequential steps pictorially. It helps the user to identify more easily the block of codes, choices and paths of execution as compared to the algorithms This is one of the major advantages of using flow charts. Table 2.1 shows the common symbols used in flow charts. The use of symbols is illustrated in the following example.

Table 2.1 : Symbols used in Flow charts

Symbol	meaning	Explanation
	Start/stop	start and end commands.
	Processing	operations are written within this symbol
	Input/output	reading and writing operations are written within this symbol
	decision	Control operations are indicated within this symbol.
	Connector	one part of the flow chart is connected to another using this symbol

	arrow	Flow of control
	Group instruction	Groups of steps or instructions
	Sub routines/ procedure	sub routine or an activity carried out as part of a function.

Example 2.2 :

The Flow chart in Fig 2.3 shows, how the mathematical expression are completed with discrete steps. Each step evaluates an expression and finally produces the result in degree Fahrenheit.

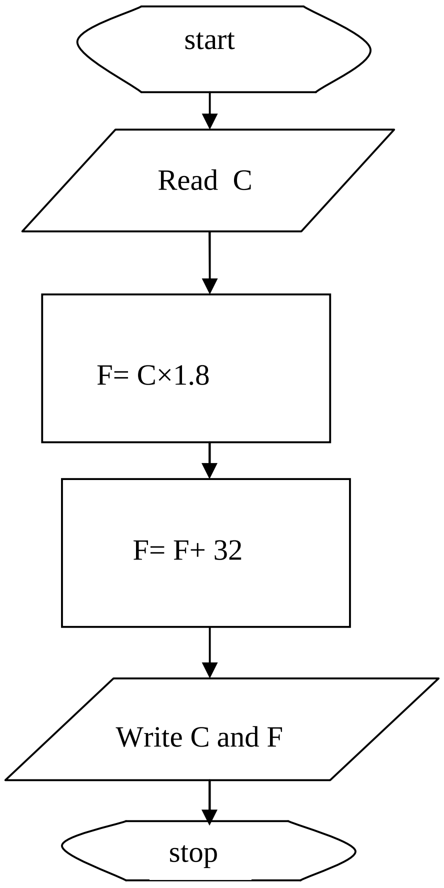


Fig. 2.3 Flow chart for converting degree centigrade into Fahrenheit

2.4 Advantages and draw backs of Flow Chart.

The use of flow charts allows us to draw a picture of the way a process actually works so that we can understand the existing process and develop ideas about how to improve it. Using a flow chart has a variety of benefits:

1. Effective analysis of the program
2. Focus in logic
3. Better way of communication: communicating the logic of a system to all concerned.
4. Proper documentation and debugging.
5. Efficient coding: conditional statements are easy to analyze.
6. Compact representation and efficient program maintenance

The major draw backs being:

1. Complex logic.
2. If alterations are needed we have to redraw the flow chart completely.
3. Reproduction: Since flow chart symbols cannot be typed, reproduction of flow chart creates problems.
4. For long flows, tracking of flow of control creates errors therby causing errors in problem solving.

2.5 Summary:

1. An a**lgorithm** is a precise set of rules / precise specification of a sequence of instructions to be

followed in solving problems using a computer.

2. The important features of an algorithm Are: (1.) definiteness, (2). Effectiveness, (3).Finiteness, (4). Input, and (5) output.

3. A flow chart is a pictorial or graphical representation of the steps necessary to solve a problem,

Unit 3: Overview of C

Structure

3.1. History of C

C is a structured general purpose machine Independent high level programming language developed by Dennis Ritchie at AT & T's Bell Labs of USA in the mid 1970s for the Unix based operating system. Many of the important concepts of C are borrowed from the language BCPL (Basic Combined Programming Language), developed by Martin Richards in 1967. Although originally designed as a systems programming language, C has proved to be a powerful and flexible language that is used for a variety of applications for nearly every available platform. The merit of C lay in the fact that it is easier to read, more flexible and more efficient at using memory. It is particularly popular for personal computer programmers because it requires less memory than other languages. C is the archetype or original model for many modern languages as when we find Language constructs in C, such as "if" statements, "for" and "while" loops, and types of variables, can be found in many later languages. Today, there are very few platforms that do not have a C compiler

In the late, seventies C began to replace the more familiar languages of that time like, ALGOL, PL/I, etc. The drawback of the B language was that it did not know data-types. Both BCPL and B are " type less" system programming languages. By Contrast, C Provides a variety of data types with powerful features. The fundamental data types are integers, characters and floating point numbers of various sizes. In addition there is a hierarchy of derived data types created with arrays, pointers, structures and union.

Since C was developed along with the UNIX operating system, it is has close association with UNIX. Major parts of the popular operating systems like windows, Linux and Unix are coded in C. This is because when it comes to performance nothing beats C. Although C is technically a high-level language, it is one of the "lowest-level" high-level programming languages in the sense; it is much closer to assembly language than are most other high-level languages. This closeness to the underlying machine language allows C programmers to write very efficient code. More over if one is to extend the operating system to work with new devices one needs to write device driver programs. These programmes are exclusively written in C.

For many years, C was the reference manual, but eventually with the appearance of many C compilers coupled with the wide popularity of UNIX operating system, it gained wide popularity among computer professionals. Today, C is the language of choice while building a variety of hardware and operating system platforms.

The American National Standards Institute (ANSI) constituted a committee in 1983, to provide an updated definition of C. The resulting definition "**ANSI C** "was completed in late 1988, and modern compilers are already supporting most of the features of this standard .The standard is based on the original reference Manual in the first edition, the classic book **"The C Programming Language"** , with little or no changes on the original design of the C language . They ensured that old programs still worked with the new standard, failing that, the compiler would produce warnings of new behavior.

One of the significant contributions of the standard is the definition of a new syntax for the defining and declaration of the function. This extra information makes it easier for compilers to detect errors caused by mismatched arguments. A second significant contribution of the standard is the definition of a library to accompany C. These library functions specifies functions for accessing the operating system, formatted input and output, memory allocation, string manipulation, and the like. A collection of standard headers provides uniform.

3.2. Importance of C

C is an immensely popular language widely used and well understood. Some of the versatile features of C language are**:** reliability, portability, flexibility, interactivity, modularity and finally efficiency and effectiveness. It is a great tool for expressing programming ideas in a way it is easily understood, regardless of the language users are most familiar with. It is in fact the original or archetypal building block for many other currently known languages and it is very close to assembly language. C is a robust language whose rich set of built in functions, and operators can be used to write any complex programs. In C large programs are divided into small programs called functions and data moves freely around the systems from one function to another. Moreover, the C compiler combines the capabilities of an assembly language with the attributes of a high level language and therefore it is useful for writing both system software and business packages without worrying about the hardware platforms where they will be implemented..The great thing about C is that it can be used to write high performance code for both application and system software. Further it can interact with hardware at quite low level. In fact, many of the compilers available in market are written in C. It is the language used for developing system applications that forms major portion of operating systems such as Windows, UNIX and Linux. C is increasingly being used in Database systems, Graphics, Spread sheets, word processors, Compilers /Assemblers, Network drivers and interpreters.

The variety of data types and powerful operators available in C makes C programs very efficient and fast. In C there are only 32 key words and its strength lies in its built-in functions. Some standard functions are available which can be used for developing programs. C Being highly portable, programs written for one computer can be made to run on another system with little or no modification.

C is at once one of the pillars of modern information technology (IT) and computer science (CS). C is a high level language that lets us to write very low level stuff like device drivers that runs as fast as assembly written programs. C's power and fast program execution come from its ability to access low level commands, similar to assembly language, but with high level syntax. It allows low level access to information and commands while still retaining the portability and syntax of a high level language. In this process C imposes few constraints on the programmer. Further it is tailor- made for structured programming, thus requiring the user to think a problem in terms of function modules or blocks. A collection of these modules make a program debugging and testing easier..Thus, C meets the requirements, where speed, space and portability are important.

Another prime feature of C is its ability to extend itself. A program in C is basically a collection of functions that are supported by the C library. We can add our own functions to the C library .With the availability of large number of functions , the programming burden becomes simple. C being simple and easy to understand, most of the operating systems and game software are written in C .

Before discussing some distinct features of C, we shall look at some sample programs in C, and as we proceed, can learn more about the language.

3.3 Sample Programs

Printing A Message: Sample program 1

The only way to learn a new programming language is by writing programs in it. Let us begin by looking at the construction of a very simple program.

The following is the output of the above program code when it is executed:

```
hello, fine
```

```
main( )
  {
   /* ……Printing begins…….*/
        Printf(“ hello, fine ”);
    /* ……Printing ends…….*/
  }
```

Fig. 3.1 The first C program to print a single line of text

In the above C program, the code begins executing at the beginning of **main**. **main( )** is a special function used by the C systems to tell the computer where the program begins. This means that every program must have a **main** somewhere. In this example, **main** is defined to be a function that expects no arguments, which is indicated by the empty list ( ). All the statements that belong to **main( )** are enclosed within a pair of braces { } as indicated above. The opening brace **"{"** indicates the beginning of the function **main** and the closing brace **"}"** marks the end of the program. All the statements between these two braces form the function body. The function body contains a set of instructions to perform the given task.

In our example, the function body contains three statements out of which only the **printf** line is an executable statement. A function is called by naming it, followed by parenthesized list of arguments, so this calls the function **printf** with the argument " hello, fine ". **printf** is a library function that prints output , in this case the string of characters (String constant or character string) between quotes.

The two lines

```
/* ……Printing begins…….*/
```

And

```
/* ……Printing ends…….*/
```

Are **comment lines** which in this program tells what the program does. Any characters between /* and .*/ are ignored by the compiler (comments are solely given for the understanding of the programmer or the fellow programmers); they may be used freely to make a program easier to understand . Any number of comments can be written at any place in the program. The normal language rules do not apply to text written with in /* and .*/ . Thus we can type this text in small case, capital, or a combination. Moreover, comment can be split over more than one line, as in,

```
/* printing
   begins.*/
```

Such a comment is often called a multi-line comment. Comments cannot be nested. For example,

```
/* Printing begins /*Printing ends.*/*/
```

Is invalid and therefore results in an error.

Let us come back to the **printf** function, the only executable statement of the program .

```
printf(" hello, fine ");
```

The above quotation can be printed in two lines, by adding another **printf** function, as in,

```
printf("hello,\n");
printf("fine");
```

The information contained between the parentheses is called the **augment** (which are simply strings of character to be printed out) of the function. The argument of the first **printf** contains a combination of two characters \ and **n** at the end of the string. The combination sequence" \n " is called **newline** and it takes the character to the next line. Therefore, you will get the output split over two lines. **\n** is one of the several Escape Sequence (similar in concept to the carriage return key on a type writer, which when printed advances the output to the left margin on the next line) available in C. if you try something like

```
printf("hello, fine
  ");
```

The C compiler will produce an error message.

No space is allowed between \ and n. **printf** never supplies a new line automatically, so several function calls may be used to build up an output line in stages, as in,

.

```
main( )
   {
    /* ……printing begins…….*/
          printf(" hello,");
         printf(" fine,");
         printf(" \n");
     /* ……printing ends…….*/
   }
```

To produce identical output. Here \n represents only a single character. An escape sequence like \n provides a general and extensible mechanism for representing hard to type or invisible characters. It is also possible to produce multi line output by one printf statement with the use of newline character at appropriate places, as in,

```
printf ("hello\n….fine,\n……I\n……..am ok!");
```

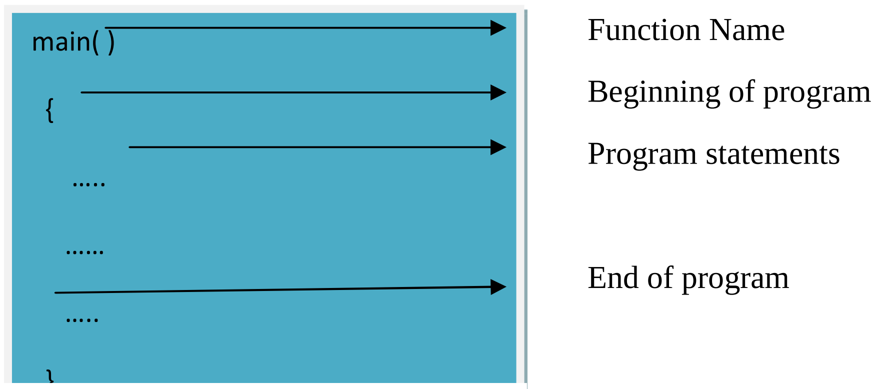
Where the output is

```
hello
 …..fine,
 ……….I
 ……….am ok !
```

The inclusion of the preprocessor directive **# include < stdio.h >** at the beginning of all programs that use any input/output library functions should not be insisted for functions like, **printf** and **scanf**, **Printf** is a pre defined standard C function (predefined in the sense that it is function that has already been written, compiled, and linked together with the program at the time of linking).

Note that the print line ends with a **semi colon**. Thus the mark **;** acts as a statement terminator. That is, every C statement must end with a **;** mark. In C , everything is written in lowercase letters. However, uppercase letters are used for symbolic names representing constants. we may also use uppercase letters in output strings like "HELLO" and "FINE".

The General format of simple C programs is shown below.



Simple C program Format

The main Function

The main ( ) is a function and is part of every program. There are different forms of main statement in C. viz.,

main ( )

int main ( )

main (void)

void main (void)

int main (void)

The empty pair of parenthesis indicates that the function has no arguments This may be explicitly indicated by using the keyword **void** inside the parenthesis. Just like the way functions in a calculator returns a value, functions in C also return a value to the operating system. That is, It is also possible to specify the keyword **int or void** before the word **main**. Some compilers permit us to return nothing or no information to the operating system from **main ( ).** In such a case we should precede it with the key word **void.** The key word **void** means that the function does not return any value to the operating system and **int** means that the function s returns an integer value to operating system. When **int** is specified, the last statement in the program must be **"return 0".**

Addition of Two numbers: Sample program 2

Consider another program, which performs addition on two numbers. This program explains the need for the use of declaration of variables, and use of operators.

/Program to add two numbers:/

```
/* addition of  two numbers */

main ( )

  {

        int  num;

        float amount;

        num = 10;

        amount = 20.25+29.85;

        printf ( " % d\n",num);

        printf ("%5.2f",amount);

  }
```

On execution of this program we will get the following output:

```
10

50.10
```

The first line of the program is a **comment line**. Comment line in the beginning give information such as name of the program, author, date etc. To indicate line numbers comment characters can also be used. in other lines. The words **num** and **amount** are variable names used to store numeric data. The numeric data may be either in **real or integer** form. In C, all variables must be declared before they are used, usually at the beginning of the function before any executable statement. The type declaration statement is written at the beginning of main ( ) function. In lines 4 and 5, the declarations

```
int  num;

float amount;
```

tells the compiler that num is an integer (**int)** and amount is a floating (**float**) point (numbers with fractional part) numbers. All declaration statements ends with a **semicolon.** The words such as **int** and **float** are called keywords and cannot be used as variable names .The range of both **int** and **float** depends on the machine you are using; 16- bit i**nts**, which lie between -32768 and +32768 , are common, as are 32-bit **ints**. A float number is typically 32-bit quantity, with at least six significant digits and magnitude generally between about 10^{-38} and 10^{+38}. While declaring the type of variable one can also initialize it as shown in line 7 and 9.That is , the statements

```
num = 10;
amount = 20.25+29.85;
```

are called the assignment statement. **Every assignment statement must have a semicolon at the end.**

The order in which we define the variables is sometimes important sometimes and sometimes not. For example,

```
int i =10, j =25;
```

is same as

```
int j= 25, i=10;
```

However,

```
float a= 1.5, b = a + 3.2;
```

Is alright. But

```
float b= a+3.2, a = 1.5 ;
```

Is not, because we are trying to use **a** even before defining it.

Moreover, the following statements would work

```
int a,b,c,d
a = b = c = d = 10;
```

However the following statement would not work

```
Int a= b= c= d =10;
```

The next statement of the program is an output statement that prints the value of **number.** The print statement

printf (“ % d\n”, num);

contains two arguments..The first argument **"%d'** tells the compiler that the value of the second argument **num** should be printed as a *decimal integer*. These arguments are separated by **comma. T**he newline character "\n " causes the next output to appear on a new line.

The last statement of the program

printf ("%5.2f", amount);

print out the value of **amoun**t in floating point format. The format specification **"%5.2f "** tells the compiler that the output must be floating type , with five places in all and two places to the right of the decimal point.

Calculation of Interest: Sample Program 3

C supports the basic four arithmetic operators (-, +, * . /) along with various others. The use of such operators along with other variable declarations, the while loop construct and # define preprocessor directive are illustrated in the program below. The program calculates the value of money at the end of each year of investment, assuming the interest rate at 11 percent with an initial investment of 50 000 for 10 years .In this program, the variable **value** represents the value of money at the end of the year and the **amount** represents the value of the money at the start of the year. The statement

amount = value ;

makes the value at the end of the current year as the value at the beginning of the *next* year .

The preprocessor compiler directive **#define,** defines a symbolic constant. Whenever a symbolic name is encountered, the compiler automatically substitutes the value associated with the name. If you want to change the value you have to simply change the definition. **#define** line should not end with a semicolon and are usually written in upper case letters(so that they can be readily distinguished from the lower case variable names), usually placed at the beginning before the **main ( )** function. They are not declared in the declaration section. The declaration section of the program declares **year** as integer and **amount ,value and rate** as floating point numbers. When two or more variables are declared in one statement, they are separated by commas. It is also possible to declare the floating point variables as multiple statements as in,

float amount;

float value;

float rate;

```
/*  .............................. INVESTMENT PROBLEM ………………….. */
# define PERIOD      10
#define   PRINCIPAL  50000.00
/* .............................. MAIN PROGRAM BEGINS ………………….. */
main (  )
 {  /*   ........................DECLARATION STATEMENTS …………….. */
          int year;
          float amount, value, rate;
/*   .............................. ASSIGNMENT STATEMENTS …………….. */
         amount = PRINCIPAL ;
         rate = 0.11;
         year = 0;
/*   ...............  ......... COMPUTATION STATEMENTS… ………….. */


/*   ...............  COMPUTATION  USING while LOOP ………….. */
         While (year < = PERIOD )
             {
               printf ( " % 2d     % 8.2 f \n" , year, amount );
               value = amount + rate * amount;
               year = year +1;
                amount = value;
             }
 /*   ...........  .....................  while LOOP ENDS… ………….. */
 }
/*   ...............  ......... PROGRAM  ENDS                  … ………….. */
```

Fig.3.5 The Investment Program

In the **while** loop all computation and printing are accomplished. The body of a **while** loop can be one or more statements enclosed in braces . The parenthesis after the **while** contain a condition that is tested. So long as this condition remains true all , all statements within body of the while loop keep getting executed repeatedly. When the condition becomes false , the control passes to the first statement that follows the body of the **while** loop..In this case as long as the value of the **year** is less than or equal to the **PERIOD**, the four statements grouped by braces that follows the **while** are executed. The loop ends when year becomes greater than **PERIOD.**

Sample Program 4: Use of Sub routines:

A very simple program that explains the use of **mul ( )** function is shown below. It uses a user defined

```
/*   ............................ PROGRAM USING FUNCTION ………………….. */

int mul  ( int a, int b);           /*   DECLARATION….. */

/*   ............................ MAIN PROGRAM STARTS………………….. */

        main ( )

          {

               int a, b,c;

               a =7;

               b =10;

               c = mul (a,b);

               printf ( “multiplication of %d and % d is  % d”, a,b,c);

           }

/*   ............................ MAIN PROGRAM ENDS

                              MUL FUNCTION STARTS………………….. */

       int mul (int x, int y)

       int p;

       {

              p = x * y;

              return ( p);

        }

/*                         MUL  (  )  FUNCTION ENDS                        . */
```

function equivalent to subroutine in **FORTRAN** or Sub program in **BASIC**. The Execution of the program will print the output

Multiplication of 7 and 10 is 70

The **mul ( )** function multiplies the value of variables x and y and the result is returned to the **main ( )** function when it is called in the statement

c **= mul (a,b);**

The **mul ( )** function has two arguments x and y (declared as integers) and when called the values of a and b are passed onto x and y respectively. This example also shows a bit more of how **printf** works.

Sample Program 5: Use of Math Functions:

There are many occasions where we often use standard mathematical functions like cos, sin, exp, etc.

```
/*   ... PROGRAM USING COSINE FUNCTION …………….. */

# include < math.h >

# define PI  3.1416

# define MAX 180

main ( )

  {

      int angle;

     float x,y;

     angle = 0;

     Printf ( “Angle      Cos(angle) \n\n “);

     While (angle < = MAX)

       {

            x = ( PI/MAX) * angle;

            y = cos (x);

            printf ( “% 15 d % 13.4 f\ n “, angle, y);

            angle = angle +10;

        }

  }
```

The standard mathematical functions are defined and kept as a part of **C math library** for use in programs. The use of any of these mathematical functions in the program can be accomplished by means of **# include** instruction in the program. The **#include** directive tells the preprocessor to treat the contents of a specified file as if those contents had appeared in the source program at the point where the directive appears Like **# define**, it is also a compiler directive and tells the compiler to link the specified mathematical functions from the library. The instruction is of the form

# include < math.h >

math.h is the file name containing the required information. Program code,(Figure 3.1) explains the use of cosine function. Another # include instruction that is often used is

# include <stdio.h>

<stdio.h> refers to the standard I/O header file containing standard Input output functions. That is, it adds the contents of the file named **stdio.h** to the source program and the ankle brackets cause the preprocessor to search the directories specified by the Include environment variable for stdio.h, after searching directories specified by the / I compiler option. For example, to use the function printf( ) in a program, the line

#include <stdio.h>

Should be at the beginning of the source file, because the definition for printf() is found in the file stdio.h.

As explained earlier, C programs are divided into modules or functions. To use any of the standard functions, the appropriate header file should be included...Header files contain definitions of functions and variables which can be incorporated into any C program by using the pre-processor *#include* statement. This is done at the beginning of the C source file . To access the functions stored in the C library, it is necessary to tell the compiler about the files to be accessed. This is achieved by the use of pre processor directive

#include <filename>

Placed at the beginning of the program. Note here that **filename** is the name of the library file that contains the required function definition.

3.4 Basic Structure Of C Programs

The programs in C so far discussed illustrates that it can be viewed as a group of building blocks called functions. A function is a segment that groups a number of program statements to perform specific task. To write a c program , we must first create functions and then put them together.

The different sections of a C program as shown in figure 3.2..The documentation section consists of a set of comment lines giving the name of a program, author, date and other details, which the programmer would like to use later .The link section provides instructions to the compiler to link functions from the system library. All symbolic constants are defined in the definition section. Global variables (variables that are used in more than one function) and all the user defined functions are declared in the global declaration section that is out side of all the functions.

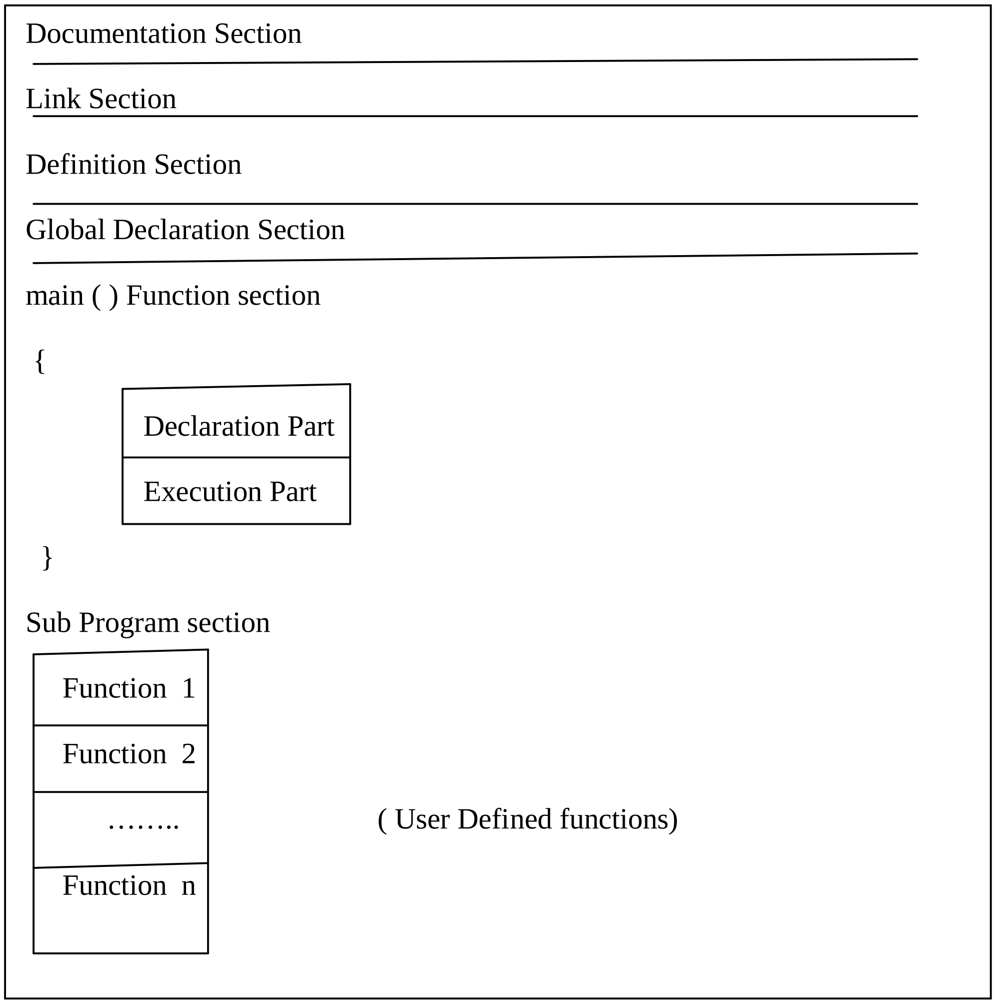


Fig.3.2 An over view of C program

Every C program must have one main ( ) function section that contains two parts, the declaration and executable part, appearing between the opening and closing braces. In the declaration part all those variables used in the executable part are declared..There is at least one statement in the executable part. The program execution begins at the opening brace and ends at the closing brace which marks the logical end of the program. Every statements in the declaration and executable parts end with a semi colon **(;)**.\

The sub program section contains all the user defined functions that are called in the **main** function. User defined functions are generally placed immediately after the **main** function, although they may appear in any order. All sections , except the main function may be absent when they are not required.

3.5 Programming Style

Programming style is a set of rules or guidelines used when writing the source code for a computer program. It is often claimed that following a particular programming style will help programmers to read and understand source code conforming to the style, and help to avoid introducing errors.

C has no specific rules for the position at which a statement is to be written. That's why it is often called a free –form language. First of all, all statements are entered in small case letters. Upper case letters are used only for symbolic constants. The statements in the program must appear in the same order in which we wish to be executed.; unless of course the logic of the problem demands a deliberate "jump", which is out of sequence. These statements are terminated with a semi-colon **(;)**, and are collected in sections known as functions. By convention, a statement should be kept on its own line. Blank spaces may be inserted between two words to improve the readability of the statement. However , no blank spaces are allowed with in a variable, constant or key word.

Since C is a free-form language, we can group statements together on one line. The statements

```
a = b;
x = y-1;
z = a-1;
```

can be written on one line as

```
a = b;  x = y-1; z = a-1;
```

The program

```
main ( )
{
    Print f ("hello");
 }
```

May be written in one line like

```
main ( )  { Print f ("hello")};
```

However, this style makes the program more difficult to understand. Rather than putting everything on one line, it is much more readable to break up long lines so that each statement and declaration goes on its own line.

Comments in code can be useful and they provide the easiest way to set off specific parts of code (and their purpose); as well as providing a visual "split" between various parts of your code. Having good comments throughout your code will make it much easier to remember what specific parts of your code do. Care should be taken not to over emphasize generous use of comments inside the code. For debugging as well as testing of the code Judiciously inserted comments is very helpful and it improves the code readability as well as the understandability of the code logic.

3.6 Executing A C Program

C program Execution involves the following steps

1. **Creating the program**
2. **Compiling the program**
3. **Linking the program with functions that are needed from the C library**
4. **Executing the program.**

Although these steps remain the same irrespective of the operating system, system commands for implementing the steps and conventions for naming the files may differ on different systems. An operating system is a program that controls the entire operations of a computer system. All I/O operations are channeled through the operating system. It is an interface between the hard ware and the user. The most popular ones today are UNIX and MS-DOS .Figure 3.10 illustrates the steps involved in the execution of C program.

3.7Unix System: Creating the program

Once you have written the program you need to type it and instruct the machine to execute it. Once we

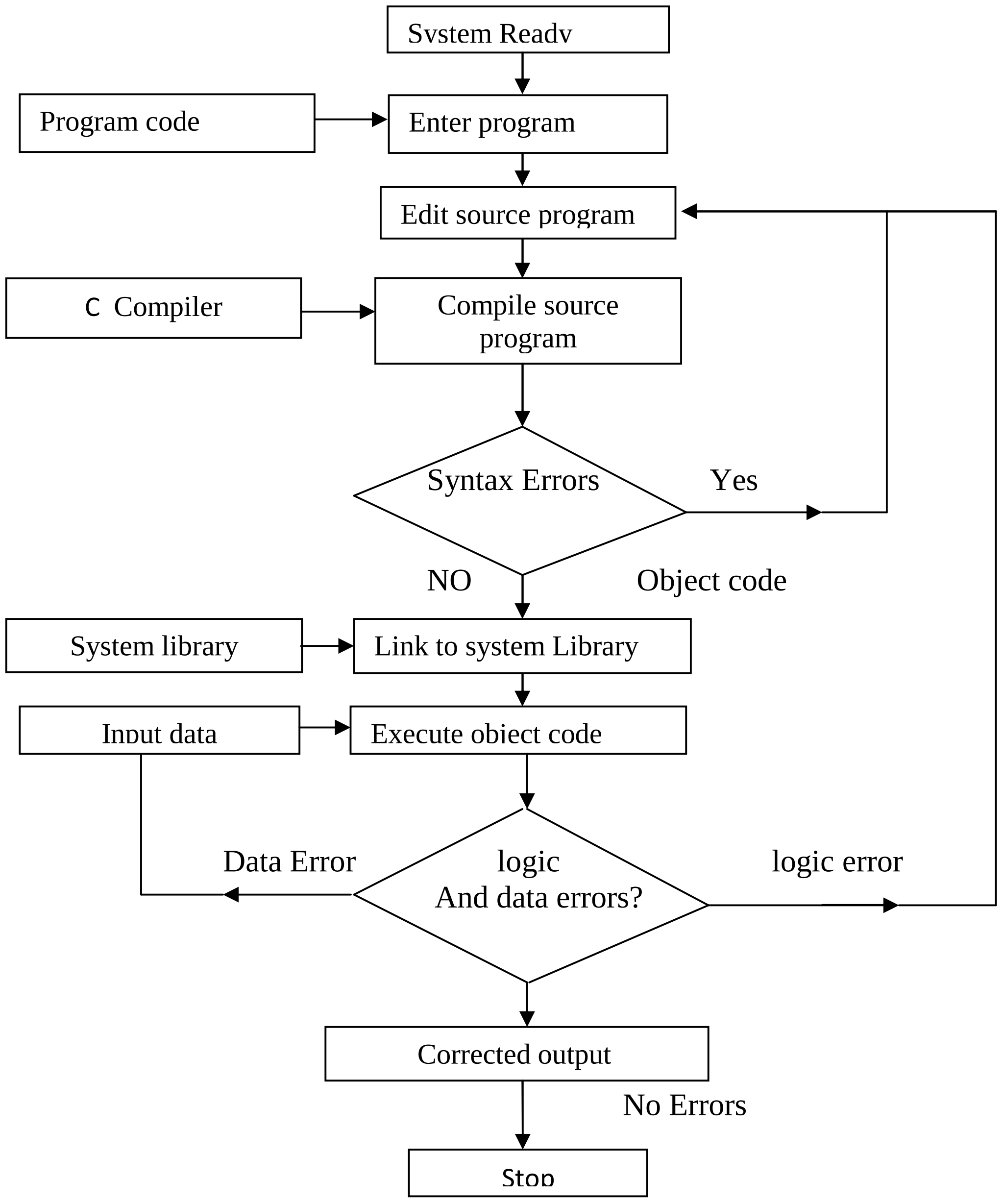


Fig.3.10 Process of compiling and running in C

load the UNIX OS in to the memory , the computer is ready to receive the program. The program must be entered into a file. The file name can consists of ,letters, digits and special characters followed by a dot and a letter c.

For eg,

hello.c

The file is created with the help of another program called text editor., either ed or vi. The command for calling the editor and creating the file is

ed filename

if the file existed before , it is loaded up. If not the file has to be created so that it is ready to receive the new program. Any corrections to the program are done under the editor. when the editing is over it is saved on the disk .It can the be referenced at any time later by its file name. The program that is entered into the file is known as ***source program*** **.**A source program is a program coded in a languages other than machine language, ad it is translated into machine language before being executed.

Compiling and Linking

Once you have written the program you need to type it and instruct the machine to execute it. To type the C program you need another program called **Editor.** Once the program has been typed it needs to be converted to machine language (0s and 1s) before the machine can execute it. To carry out this conversion we need another program called **compiler.** Assume that the source program has been created in a file named kmv.c The compilation command to achieve this task under UNIX is

cc kmv.c

The source program instructions are now translated into a form that is suitable for execution by the compiler. The translation is done after examining each instruction for its correctness. If everything is alright, the compilation proceeds silently and the translated program is stored in another file with the name ***kmv.o.*** This program is called the **object code.**

Linking is the process of putting together other programs files and functions that are required by the program. Under **UNIX**, the linking is automatically done when the **cc** command is used. Errors, if any should be should be corrected in the source program with the help of **editor** and the compilation is done again..The compiled and link program is called the **executable object code** and is stored automatically in another file named **a.out.**

Executing The Program

On compiling the program its machine language equivalent is stored as an EXE file which is an executable file. The command **a.out** would load the executable object code into the computer memory and execute the instructions .During execution, the program may request for some data to be entered *through the keyboard.*

Here are the steps that you need to follow to compile and execute your C program using Turbo C or C++.

1. start the compiler at **C** > prompt. The compiler (TC.EXE is usually present in C:\TC\BIN directory).
2. Select **New** from **File menu**
3. Type the program.
4. Save the program using **F2** under a proper name(say prog.c)
5. Use **Ctrl +F9** to compile and execute the program

6 Use **Alt +F5** to view the output.

Creating your own Executable File

Note while linking, the linker always assign the same name a.out. while Compiling a new program , this file will be over written by the executable object code of the new program .To prevent this from happening , we should rename the file immediately using the command

mv a.out *name*

Or

use the cc command option

cc-o *name* source-file

This **cc command** option will store the executable object code in the file name and prevent the old file **a.out** from being destroyed..

To compile and link multiple source program files, we must append all the filenames to the cc command.

Cc filename-1.c…… filename-n.c

These files will be separately compiled into object files called

filename-i.o

and then linked to produce an executable program file **a.out .** Also it is possible to compile each file separately and link them later .The commands,

```
c c - c mod1.c
c c - c mod2.c
```

will compile the source files mod1.c and mod2.c into object files **mod1.o and mod2.o**. They can be linked together by the command

```
c c mod1.o  mod2.o
```

Further, the source and object files can be combined as

```
C c mod1.c  mod2.o
```

Here only **mod1.c** is compiled and then linked with the object file **mod2.o.** This approach helps in situations when one of the source files need to be changed and recompiled or an existing object file is to be used along with the program to be compiled.

3.8 MS- DOS System

In **MS-DOS** system, the program is created by any word processing software in non document mode and should end with the characters **” .c.** “. For example, program.c ,pay.c , etc. Then the command

```
MSC pay.c
```

Would load the program stored in the file **pay.c** and generate the object code. This code is stored in another file under the name **pay.obj**. The linking is done by the command

```
LINK pay.obj
```

Which generates the executable code. with the file name **pay.exe**. Now the command would execute the program and give the results.

3.9 Summary

1. Every C program needs a main() function.
2. The execution of a function begins at the opening brace of the function and ends at the corresponding closing brace.
3. C programs are written in lowercase letters. Upper case letters are used for symbolic and output strings.
4. Every program statement must end with a semicolon.
5. All variables must be declared for their types before they are used in the program.
6. Include header files using # include directive for reference to special names and functions that it does not define. They should not end with a semicolon. The # sign must appear in the first column of the line.
7. When braces are used to group statements, the opening brace must have a corresponding closing brace,
8. A comment can be inserted anywhere to increase readability and understandability of the program. Comments help the users in testing and debugging. Care must be taken to match the symbols /* and */

Unit 4:Constants Variables and Data Types

Structure

4.1 Introduction

To communicate with a computer we have to speak a language which the computer understands since a computer speaks in bits bytes. . This means that, English or for that matter any other natural language by them cannot be used to perform the task of communication with computer. For this we have to have a language that is close to human language and far removed from machine language. A programming language is a methodical/systematic language designed to communicate instruction to a machine, especially to a computer and it can be used to create programs that control the behavior of a machine . However, learning C as a programming language is very much like learning English language. Learning English language begins with learning first of all the alphabets, then learning how to combine these alphabets to form words, combining words to form sentences, and finally learning to combine sentences to form paragraphs. On the same analogy , Learning C is not different. Instead of straight away learning how to write programs, we must incrementally learn (1) what alphabets, numbers, and special symbols are used in C, (2) how using these alphabets, numbers and special symbols, constants, variables and keywords are constructed, and (3) finally how these are combined to form an instruction and how groups of instructions are combined in accordance with " rules for sentence building" or syntax to form a program. The steps in learning C language is depicted below in the Figure 4.1 ,

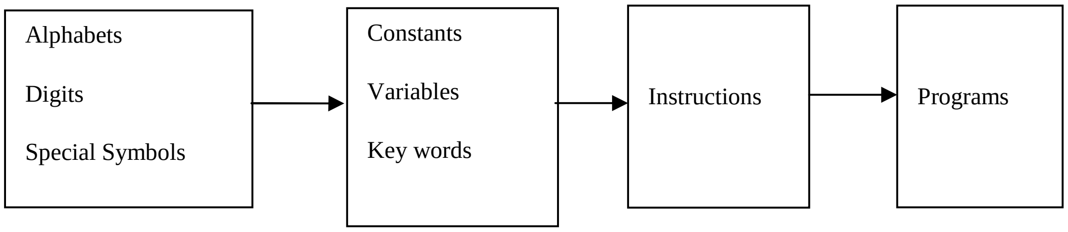


Figure 4.1: Steps in Learning C Language

As in any language, C lang ımar (*or syntax rules*) and each program instruction must conform precisely to the syntax of the language. In this chapter we will discuss the concepts of constants, variables and their types.

4.2 The C Character Set

A C character set denotes any valid alphabet, digit or special symbol, to represent an information. The set of characters that can be used to write a source program is called source character set and the set of characters available during program execution is called execution character set. Very often, in most implementations of C, both character sets are taken as identical. Generally, a character data type holds a single character(or one byte), enclosed with in single quotes, to represent a character constant. For e.g., the expressions ‘a’ , ‘b’,and ‘0’ represent character constants. Remember that “a” is used to represent a string of characters(or sequence of characters enclosed with in double quotes) and is different from ‘a’. Further, ‘\n’ is used to represent a new line character, that is used to move the cursor to a new line on the screen. Figure 4.2 shows the entire character set (i.e., the valid alphabets, numbers, special characters and white spaces) allowed in C. The compiler ignores white spaces unless they are part of a string constant. White spaces may be used to separate words, and are prohibited between characters of key words and identifiers.

Trigraph Characters

Some characters from the C character set are not available in all environments, because keyboard may not have keys to cover the entire characters set of the language. A Trigraph, is a three character replacement for a special character in the C character set. ANSI C introduces the concept of “**Trigraph**” Sequences to provide a way to enter certain characters that are not available on some keyboards. Actually, each T**rigraph** sequence contains three characters (i.e., two question marks followed by another character) as in Figure 4.3. i.e., Each trigraph sequence is introduced by two question marks followed by a third character that indicates the character to be represented. For eg., , if a key board does not support square brackets , we can still use them in a program using the **Trigraphs ?**? (and ??).

Alphabets	Upper case letters A,B,……., Z			
	Lower case letters a,b,…… .., z			
Digits	All decimal digits 0,1,2,…….9			
Special Characters	; semicolon	, comma	& ampersand	. period
	* asterisk	+ plus sign	‘ apostrophe	? question mark
	< opening bracket or less than sign	> closing bracket or greater than sign	^ caret	~ tilde
	! exclamation mark	| vertical bar	(left parenthesis	
	) right parenthesis	\ backlash	[left bracket	
	] right bracket	$ dollar sign	} right brace	
	_ under score	{ left brace	= equal sign	
	% percent sign	# number sign	/ slash	
	@ commercial at	- hyphen or minus sign	“ quotation mark	
	White Spaces			
	Blank spaces			
	Horizontal Tab			
	Carriage Return			
	New Line			

Figure 4.2 : The C Character Set

4.3 C Tokens

A **token** is a source program text that the compiler does not break down into atomic units. They are the basic building blocks/elements of the C language, constructed together to make a C program. That is, each and every smallest individual units in a C program are called **Tokens.** The **Tokens** in C language include:

1. Key words (eg: float, double etc.,)
2. Constants (eg: 100, -10.0 etc.,)
3. Strings (eg: “ABC”, “month” etc.,)
4. Operators (eg: +, - etc.,)
5. Identifiers (eg: main, total etc.,)
6. Special Symbols (eg: [ ],( ) etc.,)

C Programs are written using these **tokens** and the syntax of the language.

Trigraph Sequence	Translation
??=	# number sign
??(	[left bracket
??)	] right bracket
??<	{ left brace
?? >	} right brace
??!	| vertical bar
??/	\ back slash
??|	^ caret
??~	~ tilde

Fig. 4.3 ANSI C Trigraph Sequences

4.4 Key Words and Ident

Every C word fall under two categories, viz,. either a **key word** or an **Identifier**. C **Key words** (also called Reserved words) are the words that convey a special meaning to the C Compiler. They are the system defined **identifiers** that do have a fixed meaning (i.e., it does not change) and cannot be used as variable names. They are the basic building blocks for program statements and are written in lowercase letters. C language supports **32 (Thirty Two) keywords** and are listed in Figure 4.4.below.

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

Figure 4.4 Key words in C

An Identifier refers to the names of variables (i.e., the one which changes during program execution), names of functions, arrays, and structures. They are user defined names consisting of a combinations of alphabets, digits with a letter as the first character and underscore. The under score symbol is treated as a letter in the C character set and it helps in the readability of long variable names. That is, they are the names given to C entities such as , variables, types, functions, structures and labels in the program. However, the lengths of identifiers in C, vary from one implementation to another. In general, Identifier are created to give a unique name to C entities so as to identify it during the execution of the program. For example: **int** apple; Here apple is an identifier which denote a variable of ***integer*** type. In fact, **Keywords** (either C or Microsoft) are not used as **identifiers.**(i.e., they are reserved for special use). **Identifiers** are in general, used to name constants, functions, files and the like, apart from variables.

Rules for Identifiers

1. The first character must be an alphabet(uppercase or lowercase) or an under score.
2. All succeeding characters must be letters or digits.
3. Key words should not be used as identifiers.
4. Name of identifier is case sensitive i.e. num and Num are two different variables.
5. Identifier name cannot be exactly same as constant name which have been declared in the header file of C and you have included that header file.
6. Name of identifier cannot be exactly same as of name of function with in the scope of the function.
7. Name of function cannot be global identifier.
8. No two successive underscores are allowed.
9. Only first 31 characters are significant.
10. No special characters or punctuation symbols are used except the under score.

4.5 Constants

A **constant in C** refers to a piece of data that does not change throughout the execution of the program**. That is, C**onstants in C are expressions with a fixed value that are not changed during the execution of the program and are declared with the *define* keyword .In general, C constants can be divided into two major categories

1. Primary constants
2. Secondary constants.

These constants are further categorized as shown in Figure 4.5.

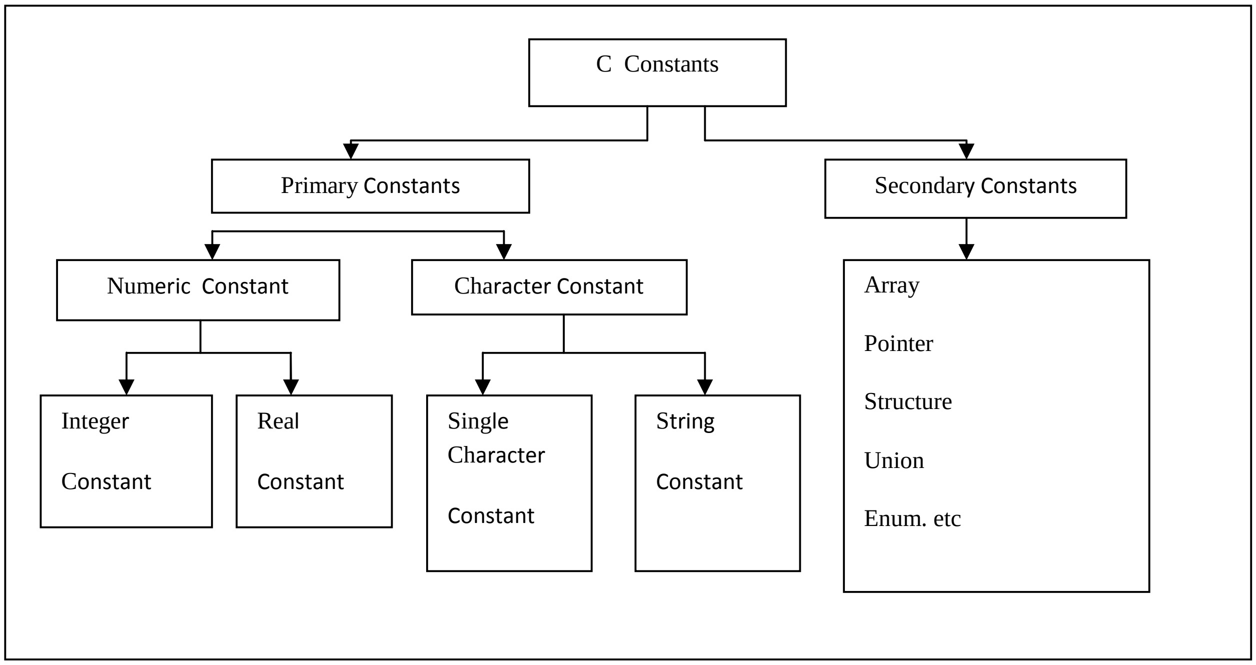


Fig. 4.5 Types of C Constants

At this stage, we would restrict our discussion to only primary constants(or basic constants) namely, Integer, Real and (of these constants..

Integer Constants

Integer constants are the numeric constants (Constants associated with number) without any fractional or exponential part. Integer constants take one of the following forms:

1. A **decimal integer.** *, e.g., 1 , 134, 10005 (* Decimal integers are a set of digits, 0 through 9, preceded by an optional – or + sign). Embedded spaces, commas, and non digit characters are not allowed between digits.
2. An **Octal integer constant** (base 8), e.g., *0 1 , 134, 0303242* . An octal constant is introduced

by a leading 0 and digits, the digits are 0 through 7 .

3. A **Hexa decimal** (base 16) Number. *e.g., 1 , 0x1, 0X186A2.* A hex constant is preceded by a leading 0X or 0x and the digits are 0 through 9 followed by A through F (Note that upper and lower case Letters are allowed) .

4. A character Constant.

Integer constants can also be suffixed with an identifier U (or u) or L (or l), which is used to indicate that the constant is unsigned or long, respectively. For e.g., 567U or 567u These suffixes may be combined *as in .e.g.,* 989712343UL or 989712343ul . The largest integer value that can be stored is machine dependent. It is 32767 on 16-bit and 2147483647 on 32-bit machines. For constructing the integer constants, certain rules have been laid down. These rules are as under:

Rules for constructing Integer constants

1.An integer constant must have at least one digit

2.It must not have a decimal point.

3.It can be either + ve or - ve.(If no sign precedes, it is assumed to be + ve.).

4. No Commas or Blanks are allowed within an integer constant.

5. The allowable range is between -32768 to 32767(For 16 bit compiler).

Real Constant

Certain quantities that vary continuously, such as prices, distances, temps, and so on, are represented by numbers containing fractional parts like 10.246. Such numbers are called **Real or Floating** point constants. That is, a real constant is one of :

- A fractional constant followed by an optional exponent
- A digit sequence followed by an exponent.

In either case followed by an optional of f, l (for single precision) , F. L(For double Precision), where:

- An optional digit sequence followed by a decimal point followed by a digit sequence.
- A digit sequence followed by a decimal point.

Further, an exponent is one of :

- E or e followed by an optional + or – followed by a **digit sequence** (A digit sequence is an arbitrary combination of one or more digits).

Floating point constants are normally represented as double precision quantities. Following rules must be observed while constructing real constants in fractional form:

1. A real constant must have at least one digit
2. It must have a decimal point
3. It could be either positive or negative
4. If no sign precedes an integer constant, it is assumed to be positive.
5. No commas or blanks are allowed within the real constant.

The exponential form of representation of real constants is usually used if the value of the constant is either too small or too large . In this form of representation, the real constant is represented in two parts. The part appearing before 'e' is called mantissa, whereas the part following 'e' is called exponent. Thus 0.000213 is represented in exponential form as 2.13e-4 . The General form is

mantissa e exponent

Following rules must be observed while constructing real constants expressed in exponential form:

1. The mantissa and exponential part should be separated by a letter e or E.

2. The mantissa part may have + ve or –ve sign.(default sign is positive).

3. The exponent must have at least one digit , which must be a +ve or _ve integer. Default sign is +ve.

4 .Range of real constants expressed in exponential form is -3.4e38 to 3.4e38.

Character Constant

Character constants are the constant which use single quotation around characters. example, \`b\`, \`k\`, \`l\` etc. In general, A character constant is a single alphabet, a single digit, or a single special symbol enclosed with in single quotes(or inverted commas). For both the inverted commas(single quotes) should point to the left. For example, 'C' is a valid character constant while ' C' is not. In C, characters are small integers, so you can use a character constant anywhere you can use an integer constant and *vice versa*. More over, the maximum length of a character constant can be 1 character.

String Constants

It is a collection of characters enclosed in double quotes. It may contain letters, digits, special characters and blank space. Examples are:

"Hello!" "How Are You " " ? " "X "

Note that a character constant (e.g., ‘X’) is not equal to the single character string constant(e.g., “X”) . Further, a single character string constant does not have an equivalent integer value while a character constant has an integer value. More over, character strings are often used in programs to build meaningful programs. Moreover, the entity having two consecutive double quotes without any characters in between them, i.e., “ “, is called a null string. Here, the quotes acts as delimiters and are not part of the string.

Backlash character constants

Sometimes, it is necessary to use newline(enter), tab, quotation mark etc. in the program which either cannot be typed or has special meaning in C programming. Such characters with special meaning should be preceded by a backlash symbol to make use of special function of them.. The backlash (\ ) causes “escape” from the normal way the characters are interpreted by the compiler. Each backlash character constant represents one character, although they consist of two characters. These character combinations are called escape sequences. Given below (Table 4.1)is the list of special characters and their purpose .

4.6 Variables

Every language should support the basic data objects namely, variables and constants. **Variables** are memory location in computers memory to store data. To indicate the memory location, each variable should be given a unique name called **identifier**. Variable names are just the symbolic representation of a memory location. These memory locations can contain integer, real or character constants. Unlike constants that remain unchanged during the execution of program , a variable may take different values at different times during execution. Examples of variable names are : sum, count, bike, interest etc. A variable name can be chosen by the programmer in a meaningful manner so as to reflect its function. Variables are to be declared before using it in the program.

Rules for writing Variable names in C

1. Variable names can be composed of letters(upper & lower case) , digits, and underscore. There is no rule for the length of a variable. A variable name is any combination of 1 to 31 alphabets.
2. The first letter of a variable should be either a letter or an under score. Note that upper and lower case are significant
3. No commas or blanks are allowed with in a variable name.
4. No special symbol other than underscore can be used in the variable name.
5. It should not be a key word.
6. White spaces are not allowed.

Constant	*Meaning*
‘\a’	audible alarm
‘\b’	back space
‘\f’	form feed
‘\n’	new line
‘\r’	carriage return
‘\t’	horizontal Tab
‘\v’	vertical tab
‘\”’	double quote
‘\’’	single quote
‘\?’	question mark
‘\\’	backlash
‘\0’	null

Table 4.1

4.7 Data Types.

Like other computer languages, **C** supports data types namely, of **integer, character and** of **float** type. In C, all variables must be declared before they are used, usually at the beginning of the function before an executable statements. A declaration announces the properties of variables; it consists of a type name and a list of variables such as

int Celsius;

int count;

The type **int** means that the variables listed are integers. ANSI C supports three classes of data types:

1. Primary data types
2. Derived data Types
3. User defined data Types.

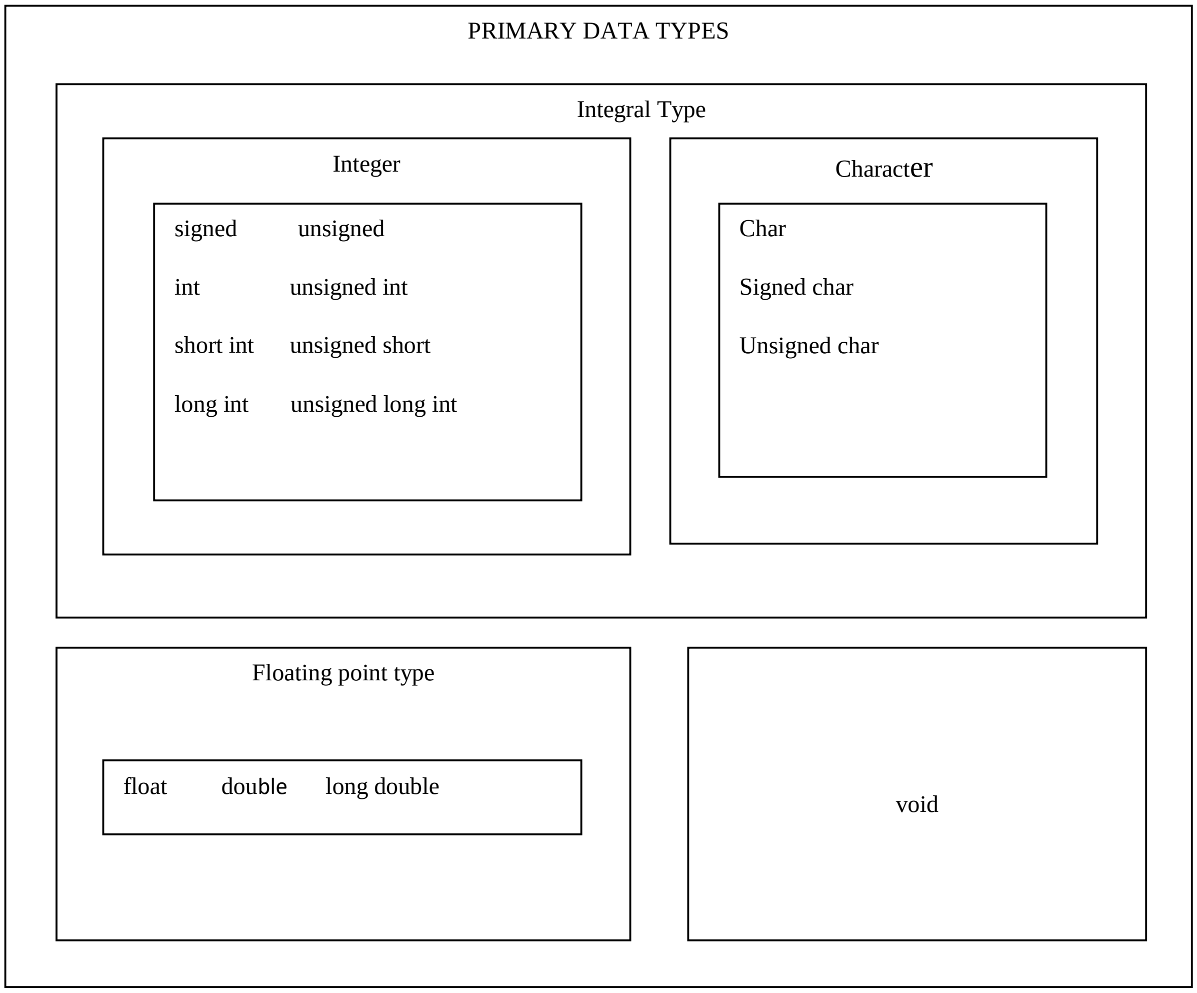


Fig. 4.6 Primary data types in C

All **C** Compilers support five fundamental data types, namely integer(**int**) , character(**char)**, Floating point(**float**), double precision floating point(**double)** and **void**. Extended data types like **long int ,long double** are also in use in C. Figure 4.6 gives an overview of primary data types in C.

Integer Types

This data type allows a variable to store numeric values. **int** keyword is used to refer integer data type. The. integers are whole numbers with a range of values supported by a particular machine (that is, the storage size of **int** data type is 2 or 4 or 8 byte. It varies with the processor in the CPU that we use). Generally, the C integer types were intended to allow code to be portable among machines with different inherent data sizes (word sizes), so each type may have different ranges on different machines. The problem with this is that a program often needs to be written for a particular range of integers, and sometimes must be written for a particular size of storage, regardless of what machine the program runs on. In fact, integers occupy one word of storage, and since the word size of machines vary, the size of integer that can be stored depends on the computer. For a 16 bit word length, the size of the integer value is limited to the range -2^{15} to $2^{15}-1$. A signed integer uses one bit for sign and 15 bits for the magnitude of the number.

In order to provide control over the range of numbers and storage space, the C language defines several integer data types: **integer, short integer, long integer, and character, all both in signed and unsigned varieties.** For eg., **Short int** represents fairly small integer values and requires half the amount of storage space as a regular **int** number uses. Unlike **signed integers**, unsigned integers use all the bits for the magnitude of the number and are always positive. To increase the range of values we declare long and unsigned integers

Floating point types

C uses the key word **float** to define floating point numbers . Floating point numbers are stored in 32-bit, with six digits precision. Key word **double i**s used to define **big** floating point numbers. It reserves twice the storage for the number. A **doubl**e data type number uses 64 bits giving a precision of 14 digits. On PC's this is likely to be 8 bytes. The **double** type represents the same data type that **float** represents, but with a greater precision. To extend the precision further, the key word **long double** with 80 bits are used.

Void types

Void is an empty data type normally used as a return type in C to declare that no value will be returned by the function. It can also play the role of generic type, meaning that it can represent any of the other standard types.

Character type

A single character of the character set of C, can be defined as a **character (or char)** type data . Key word **char** is used for declaring the variable of character type. Usually, a character enclosed between a pair of single quotes denotes a character constant. The size of **char** is 1 byte(or 8 bits of internal storage)..The qualifier s**igne**d or **unsigned** may explicitly applied to **char.**

4.8 Declaration of Variables

In order to use a variable in C, we must first declare it before they are used in the program. Declaration does two things:

1. It tells the compiler what the variable name (type name) is
2. It specifies what type of data (or properties) the variable will hold

The type declaration statement is written at the beginning of **main ( )** function.

Primary type instruction

A variable can be used to hold a value of any data type in a memory location. After assigning variable names, we have to declare them. The syntax for declaring a variable is:

data-type v1,v2,….vn;

Here **v1,v2,….vn** are the variable names and are separated by commas A declaration statement must end with a semicolon. For example,

i**nt** num, sum**;**

int code**;**

double ratio;

are valid declarations. Here, Keywords **int** and **double** are used to represent integer and real type data respectively. When qualifier is applied to the data type then it changes its size (The size qualifiers are :**shor**t and **long**) or its sign (sign qualifiers are: **signed** and **unsigned).** While using qualifiers like, **short, long, unsigned** without specifying the basic data type , the **C** compiler will treat the data type as **int** . Moreover, if we want to declare a character variable as **unsigned**, then we must do so by using both the terms like **unsigned char**

User Defined Declaration

In C language, a user can define an identifier that represents an existing data type. The user defined data type identifier can later be used to declare variables. The General syntax is:

typedef **type identifier;**

Here *type* represents existing data type and “identifier” refers to the row name given to the data type.

Example:

typedef int amount;

typedef float sum;

Here amount symbolizes **int** and sum symbolizes **float**. They can be later used to declare variables as follows:

amount dep1,dept2**;**

sum section1[20],section2[20];

Therefore dept1 and dept2 are indirectly declared as integer data type and section1 and section 2 are indirectly float data type.

Another user defined data type is enumerated data type provided by ANSI C standard which is defined as follows:.

enum identifier { value1,value2,…..valuen};

The “identifier “ here , is a user- defined enumerated data type which can be used to declare variables that can have one of the values enclosed with in the braces . After the definition we can declare variables to be of this ‘new’ type as below.

enum identifier v1,v2,…..vn;

The enumerated variables v1,v2,…vn can have only one of the values value1, value2 ….. value n.

Th assignments of the following type:

```
v1 = value3;
v5 = value1;
```

are valid.

For example:

```
enum day { Monday, Tuesday,…….,Sunday};
enum day week_ st,week_end;
week_ st = Monday;
week_end = Friday;
If (week_st = = Tuesday)
week_end = Saturday;
```

The C compiler automatically assign integer digits beginning with 0 to all the enumeration constants. That is, the enumeration constant value 1 is assigned 0, value 2 is assigned 1, and so on. The automatic assignment can be overridden if we assign enumeration constant values explicitly as;

```
enum day   { Monday = 1 , Tuesday,…….,Sunday};
```

Here Monday is assigned the value 1.The remaining constants are assigned values that increase successively by 1.

The definition and declaration of enumerated variables can be combined in one statement as in :

```
enum day   { Monday, Tuesday,…….,Sunday}  week _st, week_end;
```

4.9 Declaration of Storage Class

C has a concept of "storage class" that defines the scope and life time of variables and/ or functions within a program. Storage class specifier helps to specify the type of storage used for data objects, C program uses the following storage classes specifiers:

- auto
- register
- static
- extern

In a declaration only one storage class specifiers is permitted, as there is only one way of storing things and if the storage class specifiers in a declaration is omitted then a default is chosen, depending on whether the declaration is made outside or inside the function. For external declarations the default storage class specifiers will be *extern* and for internal declaration it will be *auto*. It is the default storage class for all local variables. The variables with local life time are allocated new storage each time execution control passes to the block in which they are defined. When execution returns, the variables no longer have meaningful values,.

r**egister** is used to define local variable (or used for variables that need quick access-such as counters) that should be stored in a register instead of RAM. The variable declared as *registe*r is stored in the CPU register, the default value of that variable is the garbage value.. That is, the variable has a maximum size equal to the register size (usually one word) and cannot have unary '&' operator applied to it (as it does not have a memory location).The scope of the variable is local to the block in which it is defined (or it contains) and the variable is alive till the control remains with in the block in which the variable is defined..

static is the default storage class for global variables. The variable that is declared as static is stored in the memory, default value of which is zero. Life of variable persist between different function calls. The *static* storage class provides a life time over the entire duration of program and are not available to the linker. Therefore, another compilation unit can contain an identical declaration that refers to different object. A *static o*bject can be declared anywhere (or it does not have to be at the beginning of the block). s*tatic* variables may be initialized in their declarations; the initializes must be constant expressions, and it is done only once at compile time when memory is allocated for the static variable. Further, the scope of the *static automatic variables* is identical to that of automatic variables; however the storage allocated becomes permanent for the duration of the program.

The *extern storage c*lass is used to give reference of a global variable or function in another file, that is visible to all program files. It is the default class for objects with file scope .The variable declared as extern is stored in the memory, the default value of that variable is being zero. Variable is alive as long as the program's execution does not come to an end . External variable can be declared outside all the functions or inside functions using 'e*xtern*' keywords. External variables may be declared outside any function block in a source code file the same way another variable is declared, by specifying the type and name(extern keyword may be omitted).Typically, when declared at the beginning of the source file, the *extern* key word is omitted. When you use *extern* the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined. If the program is in several source files and the variable is defined in several files, collect *extern* declarations of variables and functions in separate header file then included by using # include when you have multiple files and you define a global variable function which will be used in another files also then *extern* will be used in another file to reference of defined variable or function.

The extern class specifies the same storage duration as static objects, but the object of function is not hidden from the linker .Using the *extern* key word in a declaration, results in external linkage and results in static duration of the object Memory for such variables is allocated when the program begins execution, and remains allocated until the program terminates. The storage class is another qualifier(like long and unsigned) that can be used in the variable declaration as given below:

auto i**nt** count;

register char ch;

static int y;

extern long sum;

The ***extern*** and ***static*** class variables are automatically initialized to zero. ***Auto*** variables , on the other hand contain undefined (or garbage)values unless they are initialized explicitly.

4.10 Assigning Values To Variables

Variables are used in program statements. Any variable used in the program must be declared before using it in any statement. In fact, the type declaration statement is written at the beginning of main( ) function. While all the variables are declared for their type, the variables that are used in expressions (on the right side of equal sign) must be assigned values before they are encountered in the program. First we will discuss the subtle variations of the type declarations as:

(a) While declaring the type variable we can also initialize it as:

int i = 5, j = 15;

f**loa**t a = 1.2, b = 1.99;

(b) The order in which we define variable is sometimes important and sometimes not.

For e.g**., int** i = 10, j = 12; . is same as

i**nt** j= 12, i = 10.

However, **float** a= 1.5, b= a +3.2; is alright

But **floa**t b = a+ 3.2 ,a = 1.5 is not,

Because, here we are trying to use **a** even before defining it.

(c) The following statements work better

int a,b,c,d;

a = b = c = d = 10;

However the following statement would not work

int a = b = c = d = 10, an instance of using **b** (to assign to a) before defining it.

The Assignment statement

We can assign values to the variables using the assignment operator = as follows:

variable_name = constant;

Multiple assignments in one line are permitted in C. For eg.,

initial _value= 0; final value = 10; is a valid statement.

It is also possible to assign a value to variable at the time the variable is declared. This takes the following form:

data-type variable_name = constant;

More than one variable can be initialized in a single statement as:

a= b = c = 2;

x = y = z = MIN;

Note here that, **MIN** is a symbolic constant defined at the beginning.

Reading Data from Key board

There is a function in C, called the **scanf** function, which allows the programmer to accept input from the key board(or pass data to our C program). That is, Once executed our program will wait for the user inputs , once it came across any **scanf** function during program execution. It is a general input function available in **C** and is very similar in concept to the **printf** function. That is, **printf** and

scanf are two standard C programming language functions for console input and output. **scanf** works much like an INPUT statement in BASIC language. The syntax of **scanf** function is:

scanf("format string", &argument list);

The format string must be a text enclosed in **double quotes** and it contains the format of data being received for connecting it into internal representation in memory. e.g., integer (%d), float (%f), character (%c), or string (%s). The argument list contains a list of variables each preceded by the **address list** and separated by comma. The number of argument is not fixed. However corresponding to each argument there should be a format specifier. Inside the format string the number of argument should tally with the number of format specifier. For eg., if i is an integer and j a floating point number , to input these two numbers we may use s**canf("%d %f", &i, &j);.** The **& symbol** before each variable name is an **operator** that specify the variable name's address. We must always use this address. Let us look at an eg'.,

scanf("%d", &number);

when this statement is encountered by the computer, the execution stops and waits for the value of the **variable number** to be typed in. Since the control string **"%d**" specifies that it is an integer to be read from the terminal , we have to type in the value in the integer form. Once the number is typed in and the return key is pressed, the computer then proceeds to the next statement. The required header for the **scanf** function is **# include < stdio.h >.**

4.11 .Defining Symbolic Constants

A symbolic constant is a name that substitute for a sequence of characters (characters may be a numeric constant, a character constant, or a string corresponding to a character sequence) that cannot be changed..When the program is compiled, each occurrence of a symbolic constant is replaced by its corresponding character sequence compiled. They are usually defined at the beginning of the program. The symbolic constants may then appear later in the program in place of the numeric constants, character constants, etc, that the symbolic constants represent. The syntax of the Symbolic constant is:

#define symbolic- name value of constant

For example, consider a C program with the following symbolic constant definitions:

```
#define PI  3.141593
 #define TRUE 1
 #define FALSE 0
```

# define PI 3.141593 defines a symbolic constant PI whose value is 3.141593.When the program is preprocessed, all the occurrences of the symbolic constant pi are replaced with the replacement text 3.141593. Here the preprocessor statements begin with # symbol. and are not end with a semi colon. By convention preprocessor constants are written in UPPER CASE. Further during run time, the value of a symbolic constant does not change. Symbolic names are sometimes called constant identifiers. Since symbolic names are constants, they do not appear in declarations.

Rules for Symbolic Constants

1. Symbolic names have the same form as variable names written in UPPER CASE.
2. No blank space between **'#'** and the word **define.**
3. **'#'** must be the first character in the line.
4. A blank space is required between **#define** and **symbolic name** and between **symbolic name** and the **constant.**
5. **#define** (**#define** is a preprocessor compiler directive) statements do not end with a semi colon.
6. After definition, the *symbolic name* should not be assigned any other value within the program by using an assignment statement.
7. *symbolic names* are not declared for data types. Its data type depends on the type of constant.
8. **#define** statements may appear anywhere in the program but before it is referenced in the program.

4.12 Declaring a Variable as Constant

In environments that support C, we may like the value of certain variables to remain constant during Program execution. We can achieve this by declaring the variable with the qualifier *const* at initialization as in e.g.,

const int *class_size* = 20;

The **const** is a new data type qualifier defined by ANSI C. This tells the compiler that the value of the **int** variable *class_size* must not be modified by the program. However, it can be used on the RHS of an assignment statement like any other variable.

4.13 Declaring a Variable as Volatile

Although we have phrased the discussion in terms of declaring a variable as constant , by far the most frequent use of another qualifier **volatile,** that could be used to tell explicitly the compiler that a variables value may be changed at any time by any external source is imminent. For example:

volatile int date;

This means that the value of **date** may be altered by some external factors even if it does not appear on the LHS of an assignment statement. When we declare a variable as **volatile,** the compiler will examine the value of the variable each time it is encountered to see whether any external alteration has changed the value.

If we wish that the value of a variable must not be modified by the program while it may be altered by some other process, then we may declare it as both **const** and **volatile** as :

volatile const int date = 12 ;

4.14 Summary

1. The three primary constants and variable types in C are **int, float** and **character**.
2. A variable name can be of maximum 31 character.
3. Do not use a key word as a variable name.
4. Each variable used must be declared for its type at the beginning of the program or function.
5. Each variable must be initialized before they are used in the program.
6. Integer constants, by default, assume int types. To make the numbers **long** or **unsigned** , append **L** or **U** to them.
7. Floating point default to **doubl**e To make them to denote **float** or **long double** , append letters F or L to the numbers.
8. Do not use l for long.
9. Use single quote for character constants and double quotes for string constants.
10. Do not combine declarations with executable statements.
11. A variable can be made constant either by using **#define** at the beginning of program or by declaring it with the qualifier **const** at the time of initialization.
12. **'#'** must be the first character in the line
13. No blank space between **'#'** and the word **define** is allowed.
14. A **variable** defined before the main function is available to all the functions in the functions in the program.
15. A **variable** defined inside a function is local to that function and not available to other functions.
16. Input/output in C can be achieved using **scanf** ( ) and **printf(**) functions.
17. No blank space are allowed within a variable, constant or keyword.

Module II:Introduction

This is the second module of the four modules for the C programming language course in your B Sc Programme. Programming a computer at once means preparing a set of instructions for it to follow. These instructions invariably has to be written in one of the several high level languages, such as C, C++ etc. or in a low level language such as the assembly language . Eventually these instructions get translated to produce machine language program. Since it is the translated version of a program that is actually getting executed, at this level it does not matter in what language the source program may have written in. But some languages are more suited to particular applications than others: For example, C offers a mix of all the advantages of the high level languages, plus many of the desirable features which assembly alone can provide. It has a wealth of operators and library of built in functions that pave the way for easy programming. In this module you will find a quick introduction to many of the operators & functions in C managing of input and output operations. The Module consists of two units in total viz,

Unit 1: Operators and Expressions.

Unit 2: Managing Input And Output Operations.

This module starts with unit 1, in which various built –in operators are discussed in great length. Unit 2, is devoted to Managing I/O operations in C.

Unit 1:Operators And Expressions

Structure

1.1 Introduction:

C language has a wide range of built –in operators to perform various operations. The symbols which are used to perform logical and mathematical operations in a C program are called operators. These C operators are used to join individual constants and variables to frame expressions. Moreover, operators, functions, constants and variables are combined to shape expressions. That is, operators are used with operands to build expressions. For example , the following is an expression containing two operands and one operator ‘ +’ (an operator to perform addition).

8 + 5

whose value is 13. The value can be any type other than void. C offers the following operator Groups.

- Arithmetic
- Assignment
- Logical/relational
- Incremental and decrement operators
- Conditional
- Special Operators
- Bit wise operators.

1..2 Arithmetic Operators

The C arithmetic operators are the +, **-**, /, * and the modulo operator % . These C arithmetic operators are used to carry out mathematical calculations like addition, multiplication, division and modulus in c programs. Unlike /, which returns quotient, the % returns the reminder, the integer division truncates any fractional part. That is, the expression

x % y

produces the remainder when x is divided by y, and thus is zero when y divides x exactly. Note that the **operator** ‘ **%** ‘ cannot be applied on floating point or double type data. Further, C does not have an operator for exponentiation. The operators in C with their meaning are listed in **Table 5.1** below.

Integer Arithmetic

When both the operands in a single arithmetic expression are integers, the expression is called an integer expression, and the operation is called integer arithmetic. Integer arithmetic always yields an integer value. For example, for integer operands such as **a** and **b** with assigned values respectively, 15 and 5, we have:

a + b = 20

a **-** b = 10

a * b = 75

a / b = 3

a % b = 0

During integer division , if both operands are of the same sign, the result is truncated to zero. If one of them is negative, the direction of truncation is machine dependant. .That is , 6/7 = 0 and -6/-7 = 0 but -6/7 may be zero or -1(that is , machine dependent).

Similarly, **during modulo operation, the sign of the result is sign of the first operand**., as in:

-16 % 3 = - 1

-16 % -3 = - 1

16 % - 3 = 1

Operator	**Meaning**
+	Addition(unary plus)
-	Subtraction(Unary minus)
*****	Multiplication
/	Division
%	Modulo division (reminder after division)

Table 5.1 Arithmetic Operators

The Precedence to the operations associated with the operators are listed as:

Operator type	Precedence	priority
Unary Minus	1	Highest
*, / , %	2	Second
+, -	3	Third

That is, when an expression is given for evaluation, they are evaluated from Left to Right, based on the precedence associated with the operators. On the other hand, if the precedence's associated with the operators are to be overridden, it is necessary to use parenthesis in the expression. However, the expression within the parenthesis is evaluated on the basis of the precedence rule , with parentheses again evaluated from left to right. For expressions with nested parentheses, we evaluate the innermost one first, then the one immediately outside and so on.

Real Arithmetic

The C language contains the basic real arithmetic operators. An arithmetic operation involving only real operands is called real arithmetic. A real operand may accept values either in decimal or exponential form. An arithmetic operation between an integer and integer gives an integer result, while , the result of applying the real operators to real is another real. For floating point values, it is rounded to the number of significant digits permissible, and the final value is an approximation of the corrected result. For example, if operands x, y ,z are floats, then we will have,

x = 6.0/ 7.0 = 0.857143

y = 1.0/ 3.0 = 0.333333

z = -2.0 /3.0 = -0. 666667.

The operator % cannot be used with real operands

Mixed Mode Arithmetic

If operands in an expression contains both integer and real constants or variables then it is a mixed mode arithmetic expression. That is, When one of the operands is real, an operation between an integer and real always gives a **real** result. In this operation, the integer is first promoted to a real one and then operation is performed. The expression thus obtained is called a Mixed mode arithmetic expression. For e.g., 25/ 10.0 = 2.5 while, 25/10= 2.

1.3 Relational Operators

Relational operators are used to check relationship between two operands. If the relation is true, it returns value 1 and if the relation is false, it returns value zero. The relational operators are

>, > = , < , < =

They all have the same precedence. C offers six relational operators in all. These operators and their meanings are listed in Table 5.2.

Operator	**Meaning**
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
= =	is equal to
!=	is not equal to

Table 5.2 Relational Operators.

A simple relational expression contains only one relational operator . When arithmetic operations are used on either side of a relational operator, arithmetic expressions will be evaluated first and then the results are compared. Relational operators have lower precedence than arithmetic operators and are used in decision making and loops(i.e., in statements like If and while) in C programming..The Syntax Is:

ae-1 relational operator ae-2

with **ae-1 and ae-2** representing arithmetic expressions**.**

For e.g., 4.6 < = 10 TRUE

4.6 < - 10 FALSE

x+y = y+z TRUE only if sum of values of x and y are equal to the sum of values of y and z

Relational operator complements

Among the six relational operators, each one is complement of another operator. They are as:

- > is complement of < =
- < is complement of > =
- = = is complement of ! =

We can simplify an expression involving the not and less than operators using the complements as :

! (x < y)	simplified to x > = y
! (x > y)	simplified to x < = y
! (x ! = y)	simplified to x= = y
! (x < = y)	simplified to x > y
! (x > = y)	simplified to x < y
! (x = = y)	simplified to x !> = y

1.4 Logical Operators.

Logical operators are used to combine expressions containing relational operators. These operators perform logical operations on the given expressions .In C there are 3 logical operators (Table 5.3) and are:

Operator	Meaning of operator
&&	logical AND
||	logical OR
!	logical NOT

. **Table 5.3**

Logical operators perform logical-AND (&&) and logical –OR (| |) operations. Its Syntax is:

logical-AND-expression:

inclusive-OR- expression

logical –AND- expression & & inclusive- OR- expression

logical-OR-expression:

logical –AND- expression

logical -OR- expression | | logical - AND- expression

some example of usage of logical expression is:

1. If (age > 60 & & salary < 300 000)

2.If (number < 0 | | number > 1000) .

Logical operators & & and | | are used when we want to test more than one condition and to make decisions. They do not perform the usual arithmetic conversions. Instead, they evaluate each operand in terms of its equivalence to 0.The result of logical operation is either 0 or 1 and is of **int** type**.** The operands of logical-AND and logical-OR are evaluated from left to right. If the value of the first operand is sufficient to determine the result of the operation, the second operand is not evaluated . The C logical operators are described in Table 5.4 belo

Operator	**Description**
&&	If both operand are non zero logical AND produces the value 1.If either operand is equal to zero, the result is zero and if the first operand is equal to zero, the second operand is not evaluated.
||	The logical-OR performs an inclusive - OR operation on its operands. The result is 0 if both operands have 0 values. If either operands has a non zero value, the second operand is not evaluated.

Table 5.4

While using compound expressions, care should be taken in using the precedence of relational and logical operators. The relative precedence are listed as:

!	Highest
> >= < <=	
= = ! =	
& &	
| |	Lowest.

1. 5 Assignment Operators.

The assignment operators perform an arithmetic operation on the 1value and assign the result to the 1value.The usual assignment operator is the '=' ,. In addition, C has a set of less frequent *shorthand* assignment operators of the form (+ +, - =, * =, / =, % =). The syntax s;

v op = exp;

where ***v*** is a variable, ***exp*** is an expression and ***op*** is a C binary arithmetic operator.(or *short hand* binary operator). For e.g., consider the statement x + = y +1 ; this is same as x= x+(y+1). Here the operator + = means add ' y + 1 to x ' (or increment x by y + 1) . Some of the commonly used *short hand* assignment operators with their description is shown in Table 5.6. In all expressions involving these operators, the type of an assignment expression is the type of its left operand, and the value is the value after the assignment.

Statement with simple assignment operator	**Statement with assignment operator**
a = a + 1	a + = 1
a = a - 1	a - = 1
a = a* (n+1)	a* = n+1
a = a/(n+1)	a / = n+1
a = a % b	a % = b

Table 5.6. Short hand assignment operators

1.6 Increment and Decrement operators.

C provides two operators ++ and - - called increment and decrement operators and these operators are useful in controlling the loops through an index variable. The + + operator adds 1 to its operand while the decrement operator - - subtracts 1. Both of these operators are unary operators. (That is, used on single operand. ++ adds 1 to operand and - - subtracts 1 to operand respectively). For example:

Let a = 3 and b = 7

a ++ ; becomes 4 and a - - becomes 6

The unusual aspect is that ++ and - - may be used either as prefix (before the variable as in ++a) or post fix (after the variable as in a ++) . In both case effect is to increment a. But the expression ++a increments a before its value is used, while a ++ increments a after its value has been used. This means that in a context where the value is being used, not just the effect, + + a and a++ are different. For e.g., in the assignment statement x = i ++, if i =5, then x = i++ sets x= 5 , but x = ++ i sets x to 6. In both case i becomes 6. The increment and decrement operators can only be applied to variables, an expression like (i +j) ++ is illegal. In general, a prefix operator first adds 1 to the operand and then the result is assigned to the variable on the left. On the other hand, a post fix operator first assigns the value to the variable on left and then increments the operand.

Similar is the case, when we use ++ or **- -** in subscripted variable. That is, the statement

```
a[ i++ ] = 5;
```

Is equivalent to

```
a[i] =5;
i = i+1;
```

Rules for increment (++) and decrement (- -) operators.

1.They are unary operators and require variable as their operands.

2.A postfix ++ or **- -** operator used with a variable in an expression, the expression is evaluated first using the original value of the variable and then the variable is incremented(or decremented by one).

3 When prefix ++ or **- -** is used in an expression, the variable is incremented (or decremented) first and then the expression is evaluated using the new value of the variable.

4.The precedence and associativity of ++ and **- -** operators are the same as those of unary + and unary -

1.7 Conditional Operator

Conditional operator (? :) is a ternary operator (that demands three operands) consisting of symbols " **?"** and "**:** " and are used for decision making in C. The operator works by evaluating test expression, returning a value if that expression is TRUE and different one if the expression is evaluated as FALSE. The general syntax is:

identifier = (test expression) ? expression1 : expression2;

This is an expression, not a statement, so it represents a value. If the condition (or test expression) is true , it evaluates and returns expression1, otherwise it evaluates and returns expression2 .Conditional operator can be used as a short hand for some **if-else** statements. For example, consider the statements,

```
a = 10;
b = 20;
x = ( a > b) ? a : b;
```

Here in this example, x will be assigned the value of b. This can be achieved using the **if…..else** statement as follows:

```
If ( a > b)
     x = a ;
else
    x = b;
```

.1.8 Bitwise Operators

Bit wise operations in C are carried out by using operations on bits(or lowest form of data that can be accessed in digital hardware) at individual level. That means , Bit wise operators are used to perform bit operations on given two variables. Four commonly used bit wise operators in C are ~ , & ,| , and ^. Generally, Bitwise operators manipulate the value of individual bits(i.e., 1 or 0). Further, to understand "<< "and ">>" , there are two shift operators which are used to shift the position of a bit (or a set of bits) to another location, in a multi-bit value. Moreover, these operators work only on a limited number of types: **int** and **char.** That means**,** they may not be applied to data types : **float** and **double**. Bit wise operators supported by C are listed in the following **Table 5.7.**

Operator	**Description of the operator**
&	Binary AND operator copies a bit to the result if it exists in both operands(or Bitwise AND)
|	Binary OR operator copies a bit if it exists in either operand(or Bitwise Inclusive OR).
^	Binary XOR operator copies the bit if it is set in one operand but not both (or Bitwise Exclusive OR).
~	Binary Ones complement operator is unary and it has the effect of flipping bits(or Bitwise ones complement).
<<	Binary left shift operator(or bitwise left shift). The left operands value is moved left by the number of bits specified by the right operand.
>>	Binary right shift operator (or bitwise right shift). The left operands value is moved right by the number of bits specified by the right operand

Table 5.7 Bit wise operators

1.9 Special Operators

C language provides a number of special operators which have no counter parts in other languages. These operators include **comma** operator, **sizeof** operator, **pointer** operators(& and *) and member selection operator (. and **--** >) . Pointer operators will be discussed while introducing pointers and member selection operators will be discussed with structures and union. The c**omma** and **sizeof** operators are discussed in this section.

The Comma Operator

This operator is used to link the related expressions together. A coma -linked list of expressions are evaluated left to right and the value of right most expression is the value of the combined expression. For example, the statement

```
int x, y,z;
z =  ( x =10, y = 20, x * y);
```

Here the 1st statement will create three integer type variables : x, y,z . In the 2nd statement, R.H.S will be evaluated first. As a result, 10 will be stored in variable x, then 20 will be stored in variable y and then values in x and y will be multiplied, result of which will be stored in the variable z as 200 at the end of the execution. Since comma operator has the lowest precedence of all operators, the use of parentheses are necessary.

The size of Operator

The **sizeof** operator works on variables, constants and even on data types. It returns the number of bytes, the operand occupies in the memory. It is a compile time operator and when used with an operand, it returns the number of bytes occupied by its operand on that particular machine.

Examples include:

```
m = sizeof (sum);
n =  sizeof( long int);
o =  sizeof ( 235L) ;
```

The **sizeof** operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer and is also used during program execution, for dynamic memory space allocation of variables.

1.10 Arithmetic Expressions

Arithmetic expressions have numbers and variables combined with the regular numeric operators (+ , - , *, /) , as per syntax of the language and simplify to a single number .Some of the examples of C expressions are (table 5.8) given below:

Algebraic Expression	C Expression
a×b-c	a*b-c
ab/c	a*b/c
ax^2+bx+c	a*x*x+b*x+c

Table 5.8 C Expressions

1.11 Evaluation of Expression

Every expression is formed out of operands and operators. Expressions in C, are evaluated using an assignment statement of the form:

variable = expression;

Usually when a statement is encountered, the expression (on the RHS) is first evaluated and the result obtained thus, is used to replaces the previous value of the variable on the LHS. All variables used in the expression must be assigned values before evaluation is attempted. An example of a valid evaluation expression is;

x = a* b-c;

Remember that blank space around an operator is optional and adds only to improve the readability..

1.12. Precedence of Arithmetic Operators

The two distinct priority levels of arithmetic operators in C are:

* / %	High priority
+ -	Low priority

An arithmetic operation without parentheses will be evaluated from *left to right,* using the rules of operator precedence. The basic evaluation procedure involves *two left to right pass* through the expression..During the 1st pass, high priority operators (if any) are applied. and during the 2nd pass low priority operators, if any , are applied as they are encountered. For example, consider the statement,

x = a-b/3 + c*2-1

when a= 9, b=12, and c =3 , the statement becomes

x = 9- 12/3 + 3*2 -1

1st pass

Step 1: x = 9- 4 + 3*2 -1

Step 2: x = 9-4+6-1

Second pass

Step 3: 5+6-1

Step 4: 11-1

Step 5: 10

However, one can change the order if evaluation, by introducing parentheses into the expression. The same above expression in parentheses reads as:

x = 9- 12/(3 + 3)*(2 -1)

Whenever parentheses are used, the expression contained in the left most set is evaluated first and the expression on the right most the last. The steps are as follows:

First pass:

Step 1: 9-12/6*(2-1)

Step 2: 9-12/6*1

Second Pass

Step 3: 9-2*1

Step 4: 9-2

Third pass

Step 5: 7

Though the procedure here, involves three left to right passes, number of evaluation steps is equal to the number of arithmetic operators. That is, the number of evaluation steps is same (equal to 5) for evaluation without and with parentheses

It may happen that parentheses may be nested, in which case evaluation will proceed outward from the inner most set of parentheses as in eg;, x = 9- (12/(3 + 3)*2) -1 = 4 .

Rules for evaluation of Expression

1. The arithmetic expressions are evaluated from left to right using the rules of precedence.
2. When parentheses are used , the expression with in the parentheses assume highest priority
3. First parenthesized sub expressions from left to right are evaluated.
4. The precedence rule is applied in determining the order of application of operators in evaluating sub expressions.
5. The associativity rule is applied when two or more operators of the same precedence level appear in a sub expression.
6. If parentheses are nested, the evaluation begins at the inner most sub expression

1.13 Some computational problems

On most computers, any attempt to divide a number by zero will result in an abnormal termination of the program. In such instances, care should be taken to test the denominator that is likely to assume zero value so that the division by zero error may be avoided. Further, one must specify the correct type of operands and it should be of the correct range, so that any error due to over flow / under flow may be eliminated.

1.14 Type conversion in expressions

C lets mixing of constants and variables of different types in an expression. It automatically, converts any intermediate values to the proper type so that expressions can be evaluated without loosing any significance. This automatic conversion is called *implicit type conversion*. If the operands are of different types, the lower type is automatically converted to the higher type before the operation proceeds. The result is of higher type. The sequence of rules to be followed while evaluating an expression are given below.

Rules for evaluating expressions

All s**hort** and **char** are automatically converted to **int:** then

1. If one of the operand is **long double**, the other will be converted to **long double** and the result will be **long double**.
2. else, if one of the operands is **double**, the other will be converted to **double** and the result will be **double**.
3. else, if the operand is **float**, the other will be converted to **float** and the result will be **float**;
4. else if one of the operand is **unsigned long int**, the other will be converted to u**nsigned long int** and the result will **be unsigned long int**.
5. else, if one of the operands is **long int** and the other is **unsigned int**, then

(a) If **unsigned int** can be converted to **long int**, the **unsigned int** operand will be converted as such and the result will be **long int;**

(b) else, both operands will be converted to **unsigned long int** and the result will be **unsigned long int;**

6. else, one of the operands is **long int**, the other will be converted **to long int** and the result will be **long int;**
7. else, if one of the operands is **unsigned int**, the other will be converted to **unsigned int** and the result will be **unsigned int**.

Explicit conversion

Explicit conversion is used to tell the compiler to treat a variable as of a different type in a specific context. The compiler will automatically change one type of data in to another (or locally convert) to make it sense. For instance, if you assign an integer value to a floating point variable, the compiler will insert code to convert the **int** to a **float**. The **general syntax** is:

(type-name)expression

Where type-name is one of the standard C data types. The expression may be a constant, variable or an expression. Casting allows you to make this type conversion explicit, or to force it when it would not normally happen. To perform casting, put the desired type including modifiers like unsigned inside parentheses to the left of the variable or constant you want to cast. For Example

```
        float a = 5.25;
        int b = (int)a;     /*Explicit casting from float to int */
```

The value of b here is 5.

1.15 Operator Precedence and associativity

Two operator characteristics (or precedence and associatively of operators) determines how operators group with operators. Precedence is the priority for grouping different types of operators with their operands. Associativity is the left to right or right to left order for grouping operand to operators that have the same precedence. An operator's precedence is meaningful only if other operators with higher to lower precedence are present. Expressions with higher-precedence operators are evaluated first. The grouping of operands can be forced by using parentheses Operators that have the same rank have the same precedence.

For example, in the following statements, the value of `1` is assigned to both `a` and `b` because of the right-to-left associativity of the `=` operator. The value of `c` is assigned to `b` first, and then the value of `b` is assigned to `a`.

```
b = 2;
c = 1;
a = b = c;
```

Because the order of sub expression evaluation is not specified, you can explicitly force the grouping of operands with operators by using parentheses.

In the expression

```
a + b * c / d
```

the `*` and `/` operations are performed before `+` because of precedence. `b` is multiplied by `c` before it is divided by `d` because of associativity. Table 5.8 gives a complete list of C operators, their precedence levels , and their rules of association.

Operator	Description	Associativity
( ) [ ]	Function call Array element reference	Left to right Right to Left
+	Unary plus	Right to left
-	Unary minus	
++	increment	
- -	decrement	
!	Logical negation	
~	Ones complement	
*	Pointer reference	
&	address	
Sizeof (type)	Size of an object Type cast	
*	multiplication	Left to right
/	division	
%	Modulo	

+	addition	Left to right
-	subtraction	
<<	Left shift	Left to right
>>	Right shift	
<	Less than	Left to right
< =	Less than or equal	
>	Greater than	
> =	Greater than or equal to	
= =	equality	Left to right
! =	In equality	
&	Bitwise AND	Left to right
^	Bitwise XOr	Left to right
|	Bitwise OR	Left to right
&&	Logical AND	Left to right
| |	Logical Or	Left to right
?:	Conditional expression	Right to left
=	Assignment operators	Right to left
* = /= % =		
+ = - = & =		
^ = | =		
< < = > > =		
,	Comma operator	Left to right

Table 5.8 Precedence and Associativity of operators

1.16 Mathematical Functions

Mathematical functions such as cos, sqrt,log etc are frequently used in the analysis of real life problems. Most C compilers support these basic type functions. To use any of these functions in a program, we should include the line

# include stdio.h.

In the beginning of the program. Table 5.9 shows some standard mathematical functions

1.17 Summary:

1. An operator in C is used with operands to build functions.
2. Each expression in C should end with a semicolon.
3.Associativity is applied when more than one operator of the same precedence are used in an expression.
4. All mathematical functions implement **double** type parameters and return double type values.
5. On either side of binary operator, always use spaces to increase readability.
6.Care should be taken to increment/decrement operators to floating point variables.
7.Assignment =. Operator should not be confused with equality operator = = .

Function	Meaning of function
Trigonometric	
acos(x)	arc cosine of x
asin(x)	arc sine of x
atan(x)	arc tangent of x
atan2(x,y)	arctangent of x/y.
cos(x)	cosine of x
sin(x)	sine of x.
tan(x)	tangent of x.
Hyperbolic	
cosh(x)	hyperbolic cosine of x.
sinh(x)	hyperbolic sine of x.
tanh(x)	hyperbolic tangent of x.
Other functions	
exp(x)	e to the power of x.
fabs(x)	absolute value of x.
floor(x)	x rounded down to the nearest integer.
fmod(x,y)	remainder of x/y.
log(x)	natural log of x, x>0.
pow(x,y)	x to the power y.
sqrt(x)	square root of x, x > = 0.

Fig 5.9 Mathematical Functions

Unit 2:Managing Input and output Operations

Structure

2.1 Introduction

In order to learn a program effectively in C language, one should know, how to manage input and output of data to and from the screen and the key board. Most programs take some data as input and display the processed data, often as results, on a suitable medium. The two methods so far used, for providing data to program variables, rely on : (1) Assigning values to variables through assignment statements and (2) using the input function **scanf (**to read data from a key board). For getting the output results, usually the **printf** function that sends results out to a terminal, is used.

The Input and output operations are convenient for program that interact with the user, takes input from the user and print the message. Unlike, other higher level languages, C does not provide any input-output (I/O) statements as part of its syntax. Instead , a set of library functions provided by the operating system for input and output operations are borrowed and used by C. The standard library for I/O operations used in C is **stdlib**. That is , Standard input (or **stdin**) is a data stream used to receive input from user / collects characters typed at the keyboard and **stdout, is** the data stream for sending output to a device such as monitor etc., . In otherwords, to include input and output functionality in C programs, the **stdio** header is needed. Each program that uses a standard I/O function must contain the statement

# include < stdio.h >

at the beginning. This instruction tells the compiler, ‘to search for a file named **stdio.h** and place its contents at the appropriate place in the program . Indeed, the contents of the header file become part of the **source code** when it is compiled. In fact, this statement can be avoided in situations, where the functions **printf** and **scanf** have been defined as part of the C language. Here, in this chapter, a brief introduction of some common I/O function that can be used in many machines without much change is discussed.

2.2 Reading a Character

The simplest of all I/O operations is reading a character from the standard input unit(or key board) and writing it to the standard output unit(or the screen). The most basic way of reading input is by calling the function **getchar**. The C library function **getchar** gets a character from **stdin**, regardless of what it is, and returns it to the program. That is, it is used to get a character from console, and echoes to the screen. It is the most basic input function in C, included in the **stdio.h** header file. The **getchar** takes the following form:

variable_name = getchar( );

Variable name is a valid C name that has been declared as of **char** type. When this statement is encountered, the computer waits until a key is pressed and then assigns this character as a value to g**etchar** function. Since **getchar** is used on the RHS of an assignment statement, the character value of **getcha**r is in turn assigned to the variable name on the left. For example,

char = name;

name **= getchar ( );**

Will assign the character "**a"** to the variable name when we press the key **a** on the keyboard. Since **getchar** is a function, it requires a set of parentheses as shown. The use of **getchar** function is illustrated in the program (Table 6.1) below..

Program	**Output**

```
#include <stdio.h>
#include<conio.h>
int main( )
{
    char a;
    clrscr( );
    printf("Enter a character\n");
    a=getchar( );
    printf("The character entered is %c \n",a);
    getchar( );
    return 0;
}
```
	Enter a character
	b
	The character entered is b

Table 6.1: *use of* **getchar** *function*

The **getchar** function may be called successively to read the characters contained in a line of text..The following program me segment , for example, reads characters from key board one after another until the 'return key' is pressed

```
 ------------
 ------------
call  character;
character = '  ';
while ( character ! = '\n' )
   {
         character = getchar ( );
    }
 -------------
 -------------
```

The **getchar** returns the character it reads, or, if there are no more characters accessible, it will return the special value EOF ("end of file") .That is, The **getchar** function accepts any character keyed in, This includes TAB and RETURN . In other words, when we enter single character input, the newline character is waiting in the input queue after **getchar( )** returns. A dummy **getchar or fflush function (**to flush out unwanted functio**n)** may be used to get away the unwanted new line character , when we use **getchar** in a loop interactively. However, **getc** is used to accept a character from standard input.

2.3 Writing a Character

Often there do occur circumstances, where we want to solve computational problems and to display the characters therein on the console. The two special functions in C, that is designed to handle the output of character to monitor is ***putch*** and ***putchar* . That is,** Like **getchar**, there is an analogous companion C library function **putchar** that writes a single character to the standard output stream, (or console), specified by the argument **char** to s**tdout(i.e., it** is same as calling **putc(c,stdout).** The **putchar** function displays a single character on the screen. The syntax is:

putchar (variable_name);

where variable_name is a type **char** variable containing a character. **For e.g.,** the statement

```
answer = 'N'
putchar (answer);
```

will display the character N on the screen. The statement

putchar ('\n');

would cause the cursor on the screen to move to the beginning of the next line. The following example (Fig.6.1) explains the use of **putchar( )** function. ***Putch( )*** function, on the other hand is useful in writing the output, character by character, on the display.

The puts Function

The ***puts*** function stands for put string (or a bit of text) to the screen and this function works inside the main function. That means, the function puts( ) writes ***str*** to ***stdou***t, then writes a new line character. The general form of the function is:

int puts (char A [ ]);

A ***puts()*** function automatically appends a new line character at the end of any text it display and it uses a character array as parameter which is displayed on the screen. The ***puts()*** function performs a function that is similar to printf( ) with a %s conversion specifier (or formatted text display). However, **putc** is used for sending a single character to standard output.

```
# include <stdio.h>

int main ( )

  {

     char ch ;

     for (ch = ‘A’; ch < = ‘Z’ ; ch++)  {

          putchar (ch);

     }

    return (0);

 }
```

Output

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Fig.6.1 Program to read and write all the letters in English alphabet

2.4 Formatted Input

The standard formatted input function in C is **scanf (**that supply input in a fixed format**)** and is the input analog of **printf,** providing many of the conversion facilities in the opposite direction**..** The **scanf** contains two important things –the **format string** and the **address list** and it reads characters from the input file and converts them to internal form.. That is, **scanf** reads characters from the standard input, interprets them according to the specifications in format, and stores the results through the remaining arguments. Very often, This is the function used to read an input from the command line. The general format of an input statement is:

scanf(“ format string”, arg1,arg2,……, arg n);

Here the format string gives information to the computer on the type of data stored in the list of arguments arg1, arg2,….arg n and in how many columns (or address of locations) they are found. That is, format string specifies, how each input is read(.i.e., as a decimal integer, a decimal float, a character, a string and so on in matching arguments). The argument must be a pointer to a data type that is being read. In fact, format string and arguments are separated by commas.

scanf stops when it exhausts its format string, or when some input fails to match the control specification. It returns as its value the number of successfully matched and assigned input items. This can be used to decide how many items were found. On end of file, EOF is returned; note that this is different ' from 0, which means that the next input character does not match the first specification in the format string. The next call to **scanf** resumes searching immediately after the last character already converted. The format string usually contains conversion specifications, which are used to control conversion of input. The format string may contain:

- Blanks or tabs, which are ignored.
- Ordinary characters (not %), which are expected to match the next non-white space
- character of the input stream.
- Conversion specifications, consisting of the character %, an optional assignment suppression
- character *, an optional number specifying a maximum field width, an optional h, 1, or L indicating the width of the target, and a conversion character

A conversion specification directs the conversion of the next input field. Normally the result is placed in the variable pointed to by the corresponding argument. If assignment suppression is indicated by the * character, however, the input field is skipped; no assignment is made. An input field is defined as a string of non-white space characters; it extends either to the next white space character or until the field width, if specified, is exhausted. This implies that **scanf** will read across line boundaries to find its input, since newlines are white space

Inputting Integer l numbers

The field specification for reading an integer number is

% w sd

The percentage sign (%) indicates that a conversion specification follows.. **w** is an integer number specifying the field width of the number to be read and **d** the data type. For example, in the statement

scanf(“%3d %5d”, &num1,&num2);

the two variables in which numbers are to be stored are num1 and num2 and are of integer type. The input data items must be separated by spaces, tabs or new lines. A sample data line may thus be;

500   31246

The value 500 is assigned to num1 and 31246 to num2**.** Observe that the symbol & (ampersand) should precede each variable name, that is used to indicate the address of the variable name.

The **scanf** statement causes data to be read from one or more lines till numbers are stored in all the specified variable names. Also no blanks are permitted between characters in the format-string. The data type character d may be preceded by l to read long integers and h to read short integers.

Inputting real numbers

The **scanf** reads real numbers using the specification %f   for both decimal and exponential notation. The input field specification may be separated by any arbitrary blank spaces. If the number to be read is of double type, then

Program

```
 main( )
{
     float x,y;
     double p,q;
     printf("values for x and y is :\n");
     scanf("%f  %e" , &x ,&y);
     printf("\n");
     printf("x=  %f\n  y= %f\n\n",  x, y);
     printf("values of p and q is: ");
     scanf("%lf  %lf ", &p, &q);
     printf("\n\np = % .12lf \np = %.12e",  p, q);
  }
```

Output

```
values for x and y is : 12.3456      17.5e-2
x=12.345600
y=0.175000

values of p and q is :4.142857142857
                      18.5678901234567890
p= 4.142857142857
q=   1.8567890123456e+001
```

Table 6.2 : Reading of real numbers.

the specification should be %**lf.** Consider the statement

scanf("%f %f %f", &p,&q, ,&r) ;

with the data line

462.85 41.23E-1 543

It will assign the value 462.85 to p, 41.23E-1 to q and 543.0 to r. T he program (Table 6.2) below shows how to read real numbers in both decimal and exponential notation

Inputting character strings

A **scanf** function can input strings containing more than one character. The syntax is:

%ws or %wc

The corresponding arguments should be a pointer to character array. When the argument is a pointer to a ***char*** variable, then %c may be used to read a single character**.** Some **scanf** versions support the following string conversion specification.:

% [characters]

% [^ characters]

The specification **% [characters**] imply that only the characters within brackets are permissible in the input string. Any encounter of other string characters, will terminate the string. The specification % **[^characters]** does exactly the reverse. That is , characters after the ^ are not permitted in the input string, The reading of the string will be terminated at the encounter of one of these characters.

Reading Mixed data types

scanf can be used to input data containing mixed mode type. When one attempts to read an item that does not match the type , the **scanf** function does not read any further and immediately returns the value read. For e.g.,

scanf("%d %c %f", c %s " , &count, &code, &ratio, &name) ;

will read the data line

15 p 1.453 coffee

Correctly and assign values in the order in which they appear.

Rules for scanf

- Each variable to be read need a filed specification and a variable address of proper type.
- For any non -white space character used in the format string there must be a matching character in the user input.
- Ending the format string with white space will result in error.
- The **scanf** reads until:

1. A whitespace character is found in the numeric specification or
2. Maximum number of characters have been read
3. An error is detected.
4. The EOF is reached

2.5 Formatted output

Formatted output refers to an output data that has been arranged in a particular format, using certain features, that are effectively exploited to control the alignment and spacing of print-outs on the terminals.. The main output routine is **printf** , which writes a formatted string to the **stdout** stream. The **printf( )** function is used to print the character, string, float, integer, octal and hexa decimal values on to the output screen and it returns the number of characters that was written if an error occurs, it will return a negative value. The required header for the **printf function** is:

#include <stdio.h>

The general form of **printf** statement is :

printf (" control string"' arg1,arg2,…., arg n);

Control string consists of three types:

1.character that will be printed on the screen as they appear.

2.format specification

3.escape sequence characters like, \n,\t, and \n.

The control string specifies the number of arguments (or variables whose values are formatted and printed according to the specification of control string) that follow with their types. The arguments should match in number, order and type with the format specification. A simple format specification is as:

% w. p type-specifier

Where w , is an integer specifying the total number of columns for output value and p is another integer that specifies the total number of digits to the right of the decimal point or the number of characters to be printed from a string.

Printf formatting is controlled by 'format identifiers' which in the simplest form are listed below:

%d % i	decimal signed integer.
% o	octal integer
%x % X	Hex integer
%u	unsigned integer
% c	character
%s	string
%f	double
%e %E	double
%p	pointer
%n	number of characters written by this printf, no argument expected
%%	% .No argument expected.

Output of Integer Numbers

The format specification for printing an integer number is:

% w d

Where **w** specifies the minimum field width for the output and d , the value to be printed as an integer. However, if a number (right justified in the given field width with leading blanks) is greater than the specified field width, it will be printed in full, over riding the minimum specification. It is possible to force the printing to be left- justified by placing a minus sign directly after the % character. More over, it is possible to pad with zeros the leading blanks by placing a zero before the field width specifier. Here, The minus (-) and zero (0) are named as flags. For printing short integers we may specify **hd** . And for printing long integers the specifier **ld** is used in place of **d** in the format specifier. Some examples of different format are:

Format	output					
Printf(%d"', 1076)	1	0	7	6		
Printf(%6d"', 1076)			1	0	7	6
Printf(%-6d"', 1076)	1	0	7	6		
Printf(%06d"', 1076)	0	0	1	0	7	6

Output of Real Numbers:

Using the following form specification, the output of a real number may be displayed in decimal form:

% w.p f

The integer **w** indicates the number of positions that are to be used for the display of the value and the integer p represents the number of digits to be displayed after the decimal point. That is, the values when displayed, is rounded to p decimal places with right justification in the field of w columns, with leading trails and blanks. The default precision is actually 6 decimal places. The negative numbers will be printed with the minus sign and of the form [-] mmm-nnn.

A real number can be displayed in exponential form using the specification:

% w. p e

The display is of the form

[-] m.nnnne[±]xx

Where the length of the string n 's is specified by the precision p with the default precision being 6..Moreover, the field width w should satisfy the condition w ≥ p +7

and will be rounded off and printed right justified in the field of w columns. Further, padding the leading blanks with zeros and printing with left justification using flags 0 or – before the field specifier is also possible. Following are some examples:

Format output

Printf(“%5.3f”,x)

9	.	8	7	6

Printf(%5.2f”’,x)

	9	.	7	6

Printf(%-5.2f”’,x)

9	.	7	6	

Printf(% -8.2e”’,x)

9	.	7	6	e	+	0	1

For dynamic format specification during run time (i.e., with field width and precision given as arguments for w and p) we have the special field specification:

printf(“%*.*f” , width, precision, number);

For e.g.,

printf(‘%*.*f”, 7,2, number);

Is equivalent to

printf(‘%7.2f”, number);

Printing of a single character

A single character can be displayed in the keyboard at the desired position , right justified in the field of w column (with default value for w being 1) using the format

% wc

Printing of strings

The format specification for outputting strings is similar to that of real numbers.. The format being:

% w. ps

With w the field width for display and p indicates that only first p characters of the string are to be displayed with right justification..Some examples are:

Table showing specification and out put

%s (specification) output

N	E	W		D	E	L	H	I		1	1	0	0	0	1				

%20s(specification) output

				N	E	W		D	E	L	H	I		1	1	0	0	0	1

% 20.10s(specification) output

										N	E	W		D	E	L	H	I	

%.5s(specification) output

N	E	W		D															

%-20.10s(specification) output

N	E	W		D	E	L	H	I											

%5s(specification) output

N	E	W		D	E	L	H	I		1	1	0	0	1					

Mixed data output

Mixed data types in one printf statement is permitted in C. For e.g.,

p**rintf(“%d % f % s %c ,a,b,c,d);** is a valid one.

code Meaning

%c	Print a single character	
%d	Print a decimal number	
%e	Print a floating point number in exponent form	
%f	Print a floating point number	Without
%g		exponent form
%i	Print a floating point number	Either e-
% o	type	
%s		or f-
%u	type	
%x	Print a signed decimal integer	
	Print an octal integer without leading zero.	
	Print a string	
	Print an unsigned decimal integer	
	Print a hexagonal integer, without leading 0.s	

Table 6.1 printf format codes

Remember that, the format specification should match the variables in number, order and type. Table 6.1 below shows commonly used **printf** format codes

The letters used as prefix for certain conversion characters are:

h	short integer
l	long or double
L	for long double .

2.6 Summary

1. While using **getchar**, clear all unwanted characters on the console.
2. While using I/O functions always use the header < stdio.h >.
3. For functions that use character handling use the header< ctype.h>
4. For any variable to be read or printed, the proper field specification is to be done.
5. Always enclose format control strings in double quotes.
6. While using scanf the address specifier & ampersand is to be used.
7 Single character constants are to be enclosed in single quotes.
8. Avoid white space at the end of format string and use comma after he format string in **scanf** statements.
9. Do not use commas in the format string of a **scanf** statement.

Module III: Introduction

This module is designed as an introduction to control structures: branching and looping. Programming languages by default execute in sequence, line by line. This is very useful since in this mode of execution, it is done in an orderly manner . But if we need to make decisions and evaluate some input and decide which path to take depending on that input then we use Control Structures. Control Structures allow programmers, to change that default sequential execution. In most Programming Languages such as C, PHP, C++, C#, Java, JavaScript, and others, we have Control Structures. The first Control Structure we are going to talk about is the "if" and "if … else". What this structure does is to evaluate the condition of the "if" statement and determine if it's true or false; then if it's true executes the statements inside the "if" body, otherwise executes statements in the else body or continues executing the rest of the program. Actually, the flow of control in a computer program may be altered in two ways. One involves alternate paths provided by if…else or switch statements; the other is through the repetitive execution of a set of instructions. The first mechanism is called branching, the second called looping. Branching is deciding what actions to take and looping is deciding how many times to take a certain action. In the first unit of the module, you are guided through the structure of the various branching constructs like, if…else, else…if, switch etc., with sample programs. The next unit is a tour through the control structure through looping: viz, while, do…while, for(,,) loop ,and continue statement.

Unit 1:Decision Making And Branching

Structure

1.1 Introduction.

Decision making is one of the most important concepts in C programming. That is, the programs should be able to make logical decisions based on the conditions they are in. **C** language has three major decision making instructions- the **if** statement, the **if else** statement, and the **switch** statement. These statements 'control' the flow of program execution (or they specify the order in which computations are performed), and are known as **control statements**. Here we will learn each of these, and discuss their features, capabilities and applications in more detail.

1.2 Decision Making with if statement

The key word, **if** statement, is a conditional branching statement. **It,** instructs the compiler that, what follows is a decision control instruction. That is, it allows the program to select an action (i.e., a condition is evaluated, and if it is true the statement is executed, and, the program skips past it if it is found false) based upon the user's input. The condition following the keyword if is always enclosed within a pair of parenthesis. It takes the form:

If (test expression)

A decision control instruction can be implemented in C using (1) The simple if statement, (2) The if –else statement (3) nested if-else statement and (4) else if ladder.

1.3 The Simple If Statement

The general form of **if** statement looks as:

```
if (test expression)
{
    statement block;
 }
 statement –x;
```

Here the expression can be any valid expression including a relational expression. We can even use arithmetic expressions in the **if** statement. In fact a compound statement composed of several statements enclosed with in braces (braces are used to group declarations and statements together into a compound statement or block), can replace the single statement. Remember, there is no semicolon after the right brace that ends a block. If the test expression evaluates to true, then the compound statement is executed. Otherwise the control jumps to the statement following the right brace ignoring the compound statement.. ***Please do remember that in C, a non zero value is considered to be true, where as a zero is considered to be false.*** Here is a simple program (Figure 7.1) using simple ***if statement***:

```
/* Demonstration of  if  statement*/
# include < stdio.h >
# include < conio.h>
int main ( )
 {
   int number;
   clrscr ( );
   printf ( " enter a number\n");
   scanf(" %d", &number);
   If (number > 0)
         printf(" The given number is positive\n");
   getch( );
   return 0;
}
```

output

```
enter a number
5
The given number is positive
```

Fig.7.1 program for illustration of simple if statement

On execution of this program, if you type a number greater than zero, you will get a message on the

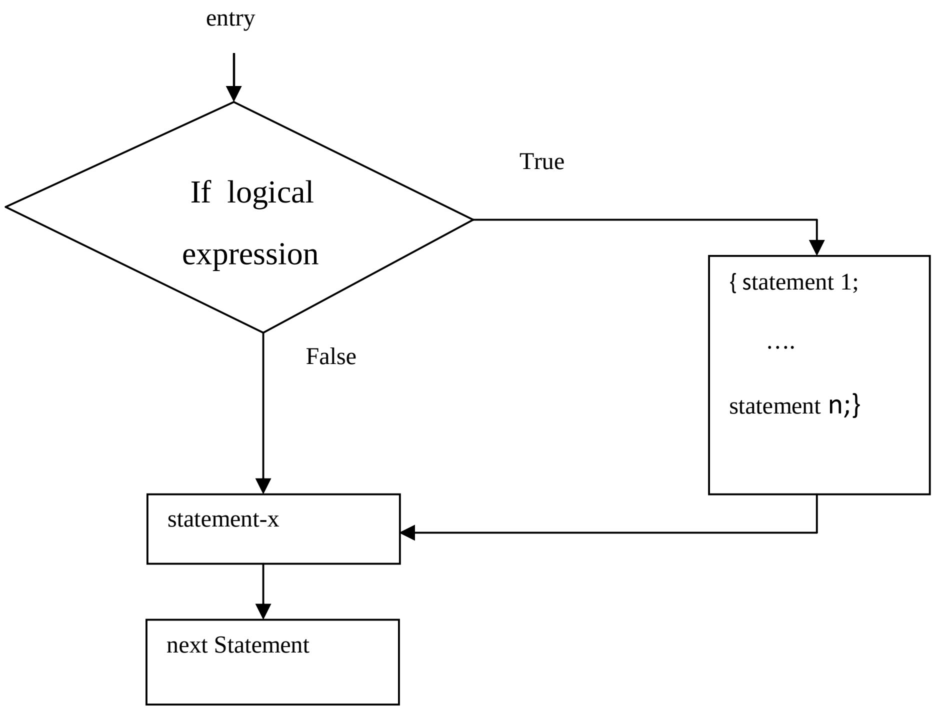


Fig.7.2 Flow chart I illustrating simple **If conditional** statement

screen through **printf( ).** If you type some other number(i.e., a number less than 0, the program does not do anything. The Flow chart given in Fig. 7.2 help you understand the flow of control in **simple if** statement.

1.4 The IF…..ELSE Statement.

The **if** statement by itself will execute a group of statements or a single statement, when the expression following it evaluates to **true** and it does nothing when it evaluates to f**als**e .In fact, the **if –else** statement is an extension of the simple **if** statement and is used to express decisions. It permits the programmer to write a single comparison, and then execute one of the two statements depending on whether the test expression (in parentheses) is true or false. That is, the **if…else** statement is used, the intention of the programmer is- to execute the group of statements denoted as true (.i.e., the true block of statements immediately following the **if** statements), or else the test expression statements denoted as false are executed..In either case, either a true or a false block of codes/statements, are executed not both .In both cases, control is transferred to the subsequent statement-x. This is interpreted in the flow chart of Fig.7.3.

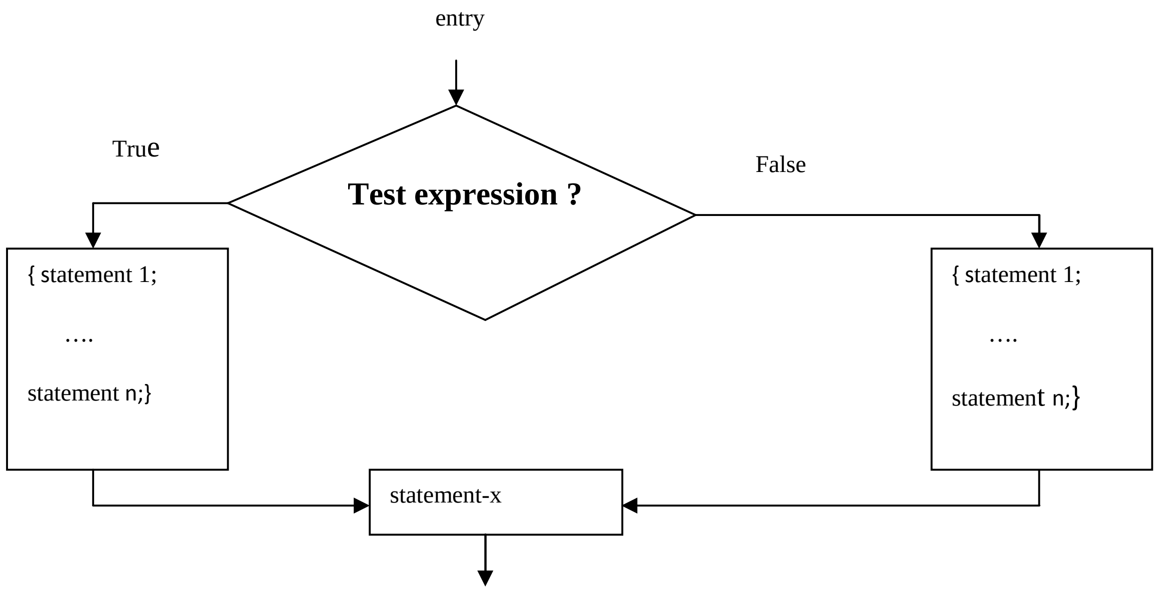


Fig.7.3 Flow chart I illustrating simple If **–else conditional** statement

Example 7.1: A program to check whether the number is odd or even?

```
# include < stdio.h >

int main ( ) {

      int number;

      printf(“ Enter a number.\n”);

      scanf(“%d”, &number);

      if ((number  % 2) = = 0)

             printf(“%d is even,” , number);

     else

            printf(“%d is odd..” , number);

           return 0;

}
```

Output

```
Enter  a number
22
22 is even.
```

Fig.7.4 A program to illustrate the If ….else statement

There are a few points that deserve worth mentioning:

1.The group of statements after the if up to and not including the else is the ‘ if block’. Similarly, the statements after the else form the ‘else block’.

2.The statements in the if and those in the else block have been indented to the right.

3. As with the if statement, the default scope of else is also the statement immediately after the else. In order to override this default scope, a pair of braces must be used.

1.5 Nested *If-else* statements.

The if….else statement can be used in nested form when a serious decision are involved. In nested if ..else construct, we write an entire if-else construct with in either the body of the if statement or the body of an else statement. The logic of execution is shown in Fig.7.5.The syntax is:

```
if  (test condition-1)
     {
       if  (test condition-2);
       {
         statement-1;
       }
       else
        {
        statement-2;
        }
   }
 else
  {
    statement-3;
   }
   statement-x;
```

Here, if the test expression -1 is false, the statement -3 will be executed; otherwise control of the program jumps to perform the second test condition. If the condition- 2 is true, the statement-1 will be evaluated, otherwise the statement-2 will be evaluated and then the control is transferred to the statement-x.

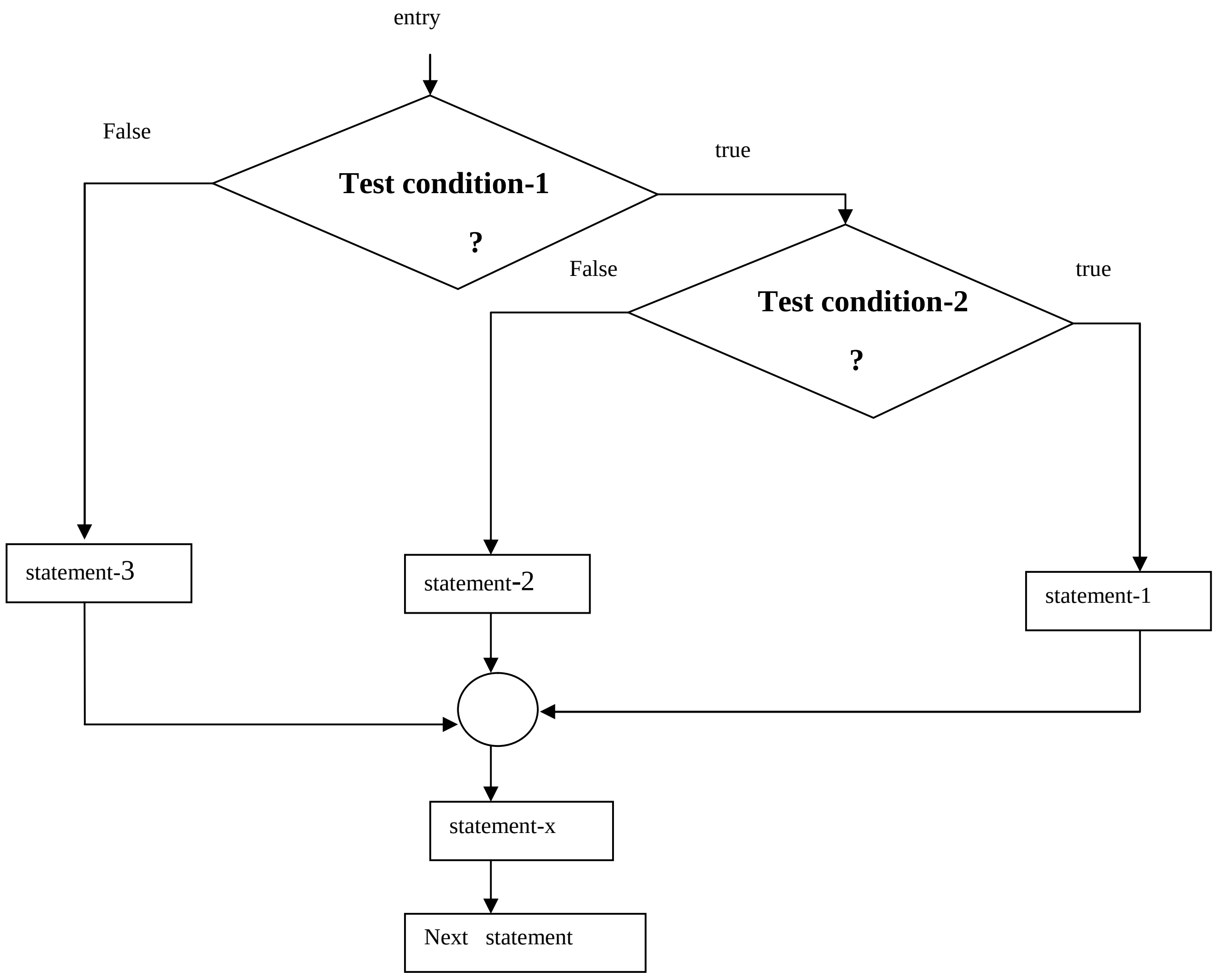


Fig.7.3 Flow chart I illustrating nested **If –else** statement

Example 7.2: A program to check whether the two numbers is <, than or > than or equal.

```
# include < stdio.h >
int main ( ) {
      int num1, num2;
      printf(“ Enter two integers.”,\n);
      scanf(“%d %d”; & num1, &num2);
      if (num1= = num2)
          printf( result: %d=%d”, num1,num2);
     else
         if(num1> num2)
             printf(“result:%d > %d”, num1,num2);
          else
             print(“result: %d >%d “,num2,num1);
 return 0;
}
```

Output

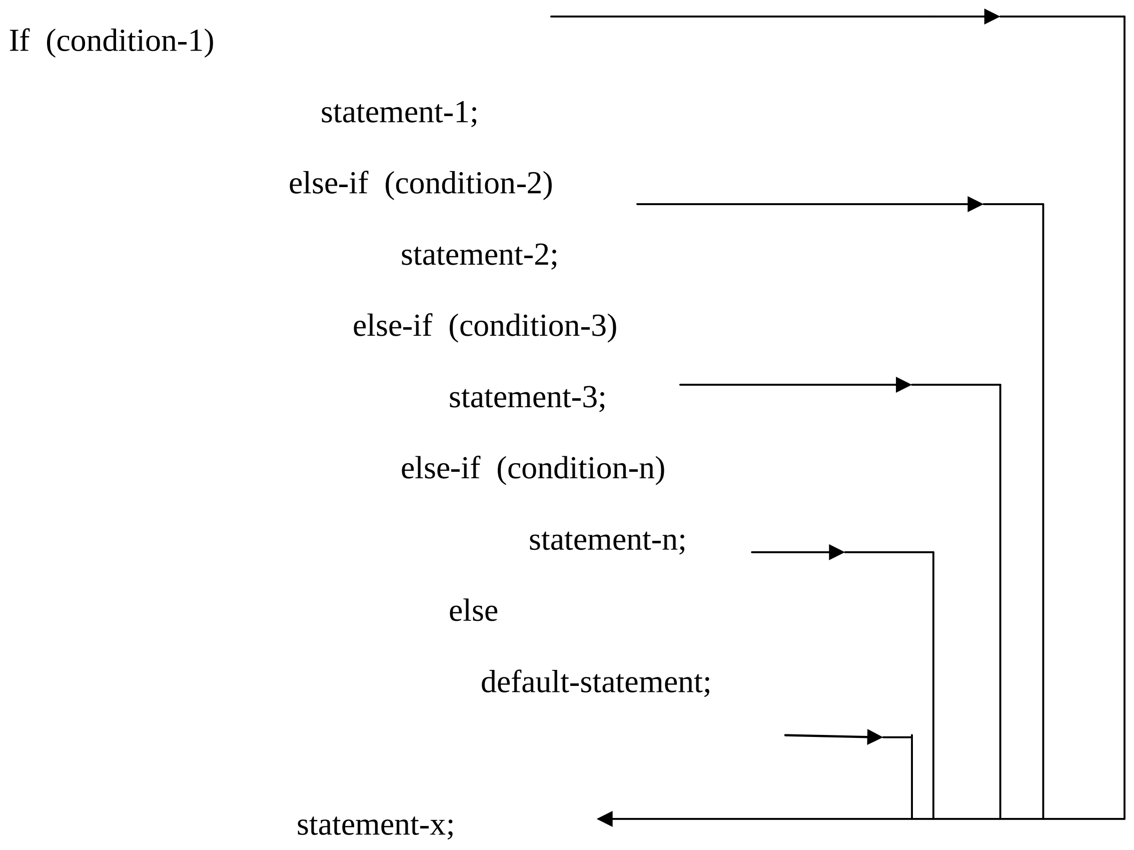
```
Enter two integers
4
2
Result:4>2
```

Fig.7.7: program illustrating **nested if -else**

1.6 The else -If Ladder

Another way of describing the nested **if-else** is the **else-if** ladder, where, every **else** is associated with an **if** statement. That is, **else-if**, is a combination of **if** and **else**. Like **else**, it extends an **if** statement to execute a different statement in case the original **if** expression is evaluated as False.

The syntax is:



```
If  (condition-1)
                    statement-1;
                  else-if  (condition-2)
                         statement-2;
                      else-if  (condition-3)
                            statement-3;
                         else-if  (condition-n)
                                statement-n;
                            else
                              default-statement;

                  statement-x;
```

This construct is called the **else-if ladder** and is useful where two or more alternatives are available for selection. In **else-if** ladder various conditions are evaluated one by one starting from top to bottom, on reaching a condition evaluating to TRUE the statement group associated with it are executed and skip other statements. If none of the expressions is evaluated to true, then the statement or group of statements associated with the final **else** is executed. In this construct nesting is allowed only in the **else** part . In fact, In **else……if** ladder, we do not have to pair **if** statements with **else** statements. That is, there is no need to remember the number of braces opened as in nested **if….else.** Moreover, **else….if** ladder produces the same effect as **nested if-else** with the benefit that it is easy to code. The flow chart corresponding to **else-if** ladder is shown in fig.7.8

In this construct, the conditions are checked, starting from the top of the **else-if** ladder, moving downwards. That is, firstly, condition-1 is checked, and if it is true, statement-1 is executed and control is transferred to statement-x. On the other hand, **If** condition-1 is false, condition-2 is checked and if true, statement -2 is executed and control is transferred to statement-x skipping the rest of the ladder .When all the n conditions are false, then the final default-statement is executed followed by the execution of statement-x. The following program(Fig.7.9) explains the **else-if** construct.

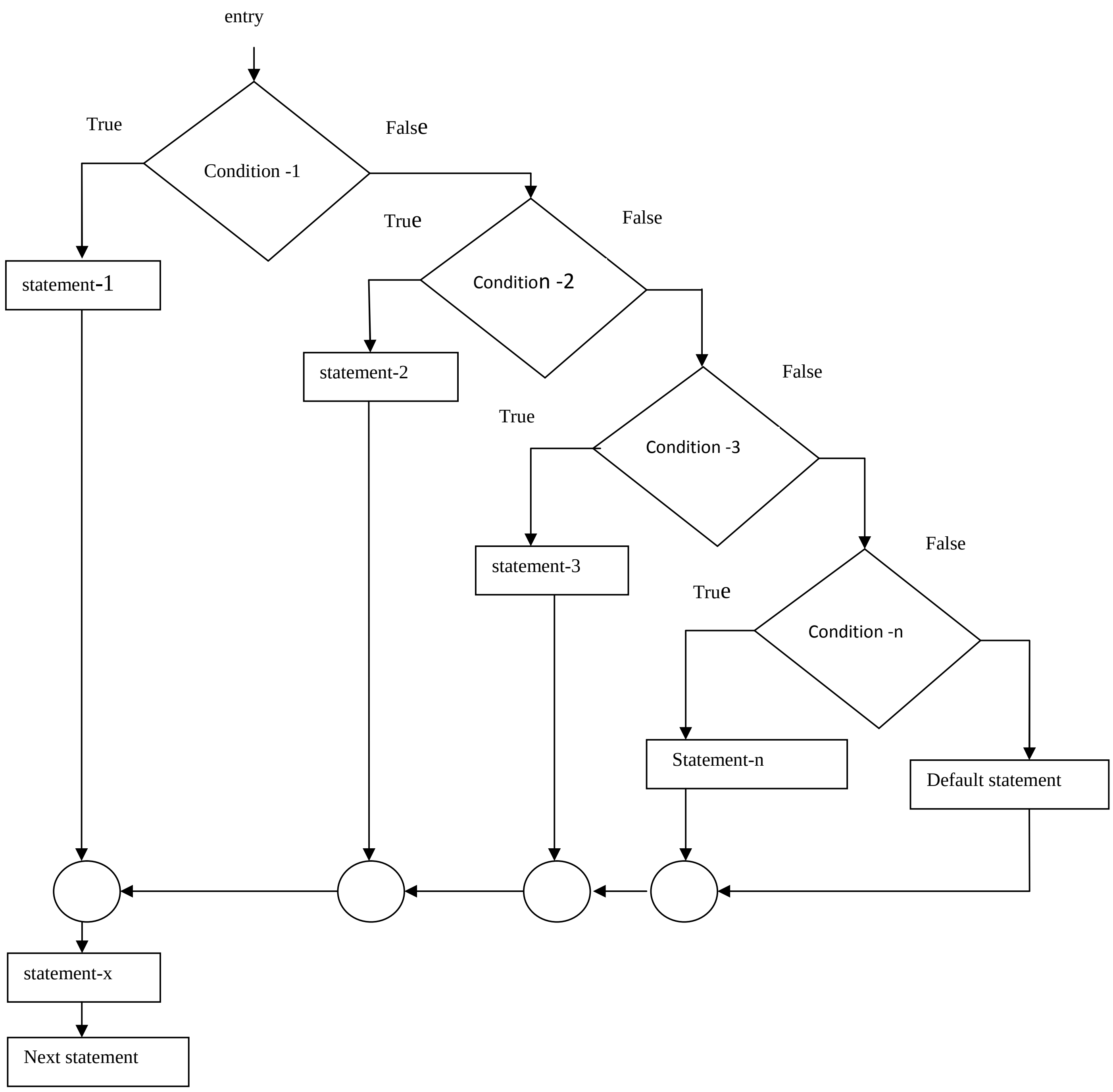


Fig.7.8 Flow chart I illustrating else- **If ladder**

```
#include < stdio.h>
#include <conio.h >
void main ( )
 {
    int num;
    clrscr( );
    printf("enter a number.\n");
    scanf("%d", &num);
    If( num = =0)
      Printf("Given number is Zero.\n");
    else if (number > 0)
      printf("Given number is positive.\n");
    else
      printf("Given number is negative.\n");
    getch ( );
   }
```

Output

```
Enter a number.
5
Given number is positive.
```

Fig.7.9. program for **else if** ladder demonstration.

Rules for Indentati

The sections of this page cover the guidelines of acceptable code indentation. Indentation is important for clarity and sticking to standard. The guidelines that are to be followed while using indentation , for control statements are listed below:

1. Indent statements that are dependent on the previous statements; provide at least three spaces of indentation.
2.Align vertically else clause with their matching **if** clause.
3.Use braces on separate lines to identify a block of elements.
4.Indent the statements in the block by at least three spaces to the right of the braces.
5.Align the opening and closing braces.
6. Indent the nested statements as per the above rules.
7. Code only one statement/clause on each line.

1.7 The Switch Statement

The switch statement is much like a nested **if** statement and it allows us to make a decision from a number of choices. In fact, it is a powerful decision making statement that allows a variable to be tested for equality against a list of values. The condition of a **switch** statement is a value. The **case** says that if it has the value of whatever is after that **case** then do whatever follows the colon. That is,.each value is called a **case,** and the variable being switched on is checked for each **switch case**. More correctly, a **switch-case default** (since these keywords go together to make up the control statement) accepts single input from the user and based on that input executes a particular block of statements. The break is used to **break** out of the case statements, and is usually surrounded by braces, which it is in. The syntax is:

```
switch   (integer expression)
  {
     case value-1;
                  block-1
                  break;
    case value-2;
                  block-2
                  break;
    …………
    ………….
    default:
                  default-block
                  break;
     }
switch
statement-x;
```

The integer expression following the key word **switch i**s any C expression that yields an integer value. It could be an integer constant or an expression that evaluates to an integer. The keyword **case** is followed by an integer or a character constant. Each constant in each **case** must be different from all the others. When the **switch** is executed, the value of the expression is compared against the values value-1,value-2,…When a match is found, the program executes the statements following that case, and all subsequent case and default statements as well .If no match is found, with any of the case statements, only the statements following the default are executed. Moreover, the **switch** statement transfers control to a statement within its body. Control passes to the statement whose **case** constant-expression matches the value of **switch (expression).** Further, execution of the statement body begins at the selected statement and proceeds until the end of the body or until a break statement transfers control out of the body. A default is optional. When present, it will be executed if the value of the expression does not match any of these **case** values .if not present, no action takes place if all matches fail and the control goes to the statement-x.

The selection process of **switch** statement is explained by the following flow diagram (Fig.7.10).

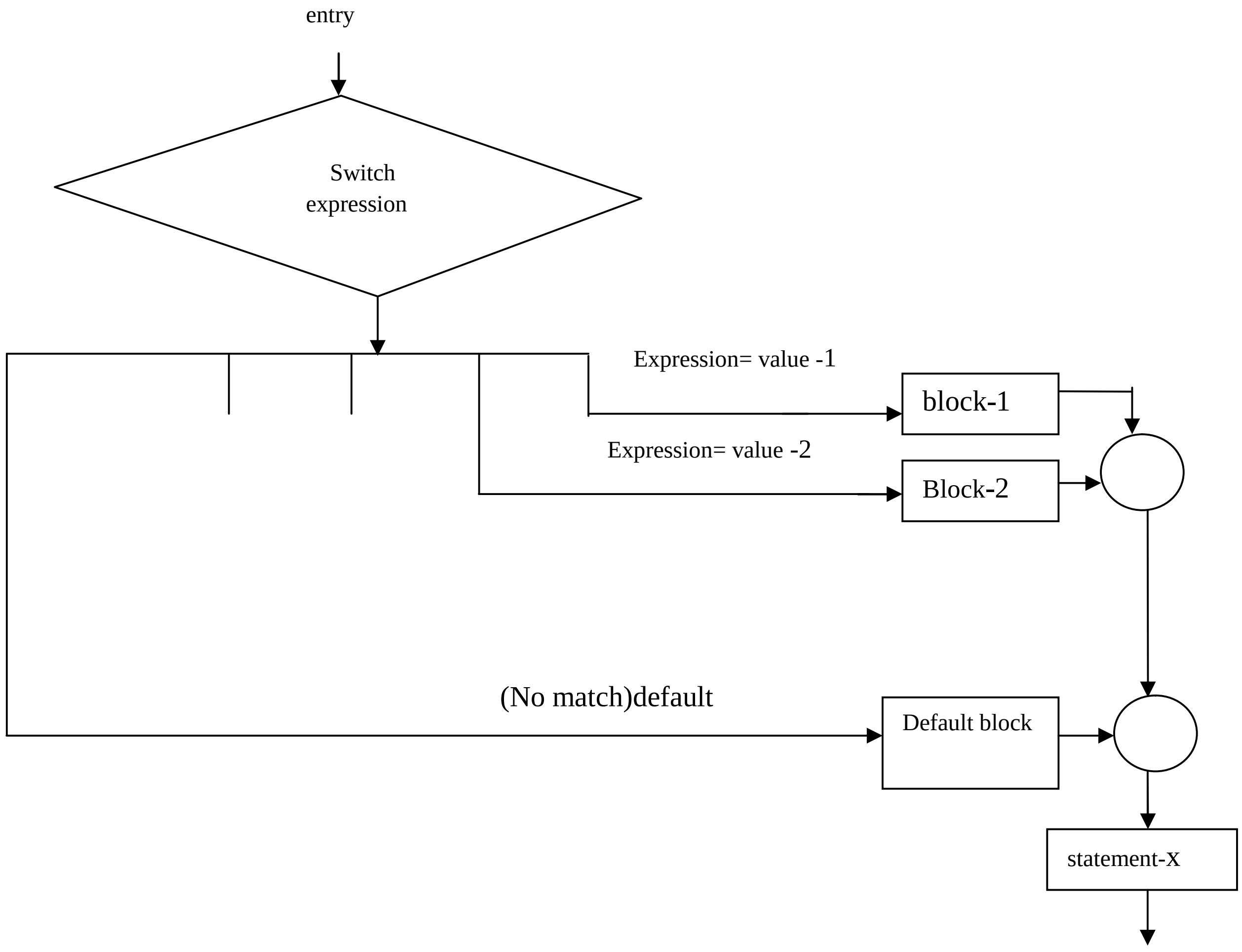


Fig.7.10 Flow chart I illustrating switch statement

The following program explain how this control structure works. Here is a program (Fig.7.11)using switch statement:

```
#include <stdio.h>

int main ( )

 {

     char grade = ‘B’;

     switch (grade)

       {

      case ‘A’ :

                Printf( “very good!\n” );

                Break;

        case ‘B’:

        case’C’ :

                Printf(“good\n”);

                Break;

       case ‘D’:

                Printf(“passed\n”);

                Break;

      case ‘F’:

               Printf(“pl try again\n”);

               Break;

      default :

               Printf(“grade invalid\n”);

      }

       Printf(“grade is %c\n”, grade);

       Return 0;

}
```

Fig. 7.11 : An example showing switch statement

This program on execution gives the following output:

Output

```
Good
Your grade is B.
```

Rules for using switch case :

1.The expression used in a **switch** statement must be an integral or enumerated type.

2.With in a **switch** statement one can have any number of **case** statements, with each **case** followed by the

v**alu**e to be compared to and a colon.

3.**case** label must be unique , and must be constants or constant expressions. case labels must end with

semicolon

4.**case** label must of integral type and should not be of floating point type.

5.When the variable being switched on is equal to a **case**, the statements following that **cas**e will execute

until a **break** statement is reached.

6.**switch** case should have at most one **default** label and can be placed anywhere in the **switch,** usually

placed at the end . d**efault** label is optional. No **break** is needed in the **default** case.

7.**break** statements takes control out of the **switc**h (or **switch** terminates and the flow of control jumps to

the next line following **switch** statement) and it is possible to share two or more case statement to

have one **break** statement.

8.Nesting(switch within switch) is permitted for **switch** statement.

9.It is not necessary that every case needs a break statement. If no break appears, the flow of control will

fall through to subsequent cases until a break is reached.

10 relational operators are not allowed in switch case statement .

1.8 The ?: Operator

The operator ?: is just like an **if..else** statement except that because it is an operator one can use it within expressions. This is a ternary operator in that it takes three values. The general form of use of this operator is:

conditional expression ? expression 1 : expression 2

Here, the conditional expression is evaluated first and the result if it is non zero, then expression 1 is evaluated and its value is returned as the value of the conditional expression. Otherwise, expression 2 is evaluated and its value is returned. For example the code segment,

```
      If (x < 0)
      flag = 0;
   else
       flag = 1;
```

can be written as

```
       flag = (x< 0) ? 0 : 1;
```

consider evaluation of yet another function

$$y = 1.5x+3 \text{ for } x \leq 2$$
$$2x + 4 \text{ for } x > 2.$$

This can be done using the conditional operator ? : as:

```
y = ( x >2)  ? (2*x+4) : (1.5 *x+3);
```

```
# include < stdio.h >

# include < conio.h >

Void main ( )

{

int a,b.c, maxm;

printf(“ program to find maxm value of three numbers:\n”);

printf(“enter the first number:\n”);

scanf(“%d”, &a);

printf(“enter the second number:\n”);

scanf(“%d”, &b);

printf(“enter the third number:\n”);

scanf(“%d”, &c);

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

printf(“the maximum  number is %d:”, maxm\n”);

}
```

Fig 7.12: illustration of the conditional operator.

.On execution of the program, the maximum variable gives the maximum value of the three numbers .

1.9 The GOTO statement

In C, GO TO statement is used for altering the normal sequence of program execution by transferring control to some other part of the program. That is ,A **goto** statement provides an unconditional jump from the **go** to a labeled statement in the function. The general form of a go to statement is:

```
   goto label;
   ………….
   …………
  label:
 Statement;
```

In this syntax label; is an identifier, to identify the place where the branch is to be made.? That is, when the control of program reaches to go to statement, it will jump to the label:, and execute the codes after it. Control may be transferred to anywhere within the current function. The **labe**l is placed immediately before the statement where the control is to be transferred. A **label:** is any valid variable name, followed by a colon and can be any where in the program either before or after the go to label; statement. During program execution when a statement like

go to begin;

Is met, the control flow will jump to the statement immediately following the label begin; This happens unconditionally.

Note that though, using **got**o statement give power to jump to any part of program, using **goto** makes the logic of the program complex and tangled .It breaks the normal sequential execution of the program. If the label: is used before the statement **goto** label; a loop will be formed and some statements will be executed repeatedly. Such a jump is called as a forward jump. On the other hand, if the label: is placed after the **goto** label; some statements will be skipped and the jump is called a forward jump.

A **goto i**s often used at the end of a program to direct the control to go to the input statement, to read further data, in fact, such goto statements puts one to enter in a permanent loop called infinite loop, until one take some special steps to terminate the program. Such infinite loops are to be avoided. Another use of goto is to transfer control out of a loop 9or nested loop) when certain peculiar conditions are encountered. Use of **goto** statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. An example to explain the control flow of goto statement is shown in fig 7, 12.Here in this program,

we want to display the numbers from 0 to 9. For this, we have defined the label statement **loop** above the **goto** statement. The given program declares a variable n initialized to 0. The **n++** increments the value of n till the loop reaches 10. Then on declaring the **goto** statement, it will jumps to the label statement and prints the value of n.

1.10 Summary:

1. There are three ways of taking decisions in a C program. - The **if** statement, the **if else** statement, and the **switch** statement. The default scope of the if statement is only the next statement.

2 An ***if*** block need not always be associated with an else block. However, an ***else*** block is always

Associated with an **if** statement.\

```
#include< stdio.h>

#include< conio.h>

int main( )

 {

  int n =0;

  loop: ;

  printf(“ \n%d”,  n);

  n++;

 if(n <10)

  {

 goto loop;

  }

getch( );

return 0

}
```

Fig.7.12 Use of *go t*o statement

3. If the outcome of an ***if else*** ladder is only one of two answers then the ladder should be replaced either with an ***else-if*** or by logical operators.

4. When we need to choose one among number of alternatives, a ***switch*** statement is used.

5. The ***switch*** key word is followed by an integer or an expression that evaluates to an integer. the case key word is followed by an integer or a character constant. the control jumps through all the cases unless the break statement is given.

6. The usage of ***goto*** is to be avoided as it obstructs the normal flow of execution.

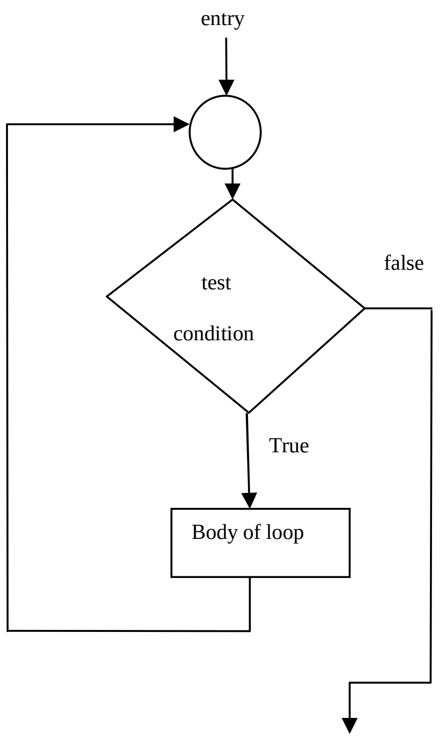
Unit 2: Decision making and looping

Structure

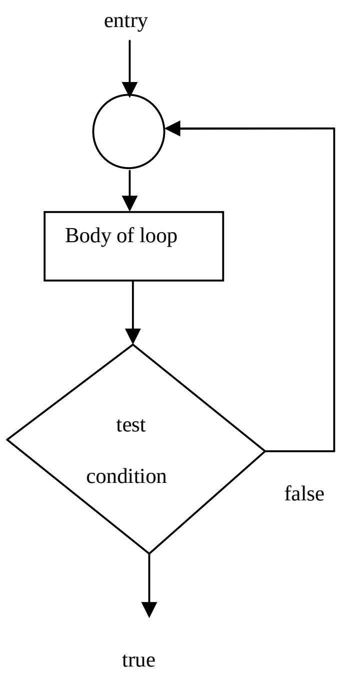
2.1 Introduction

The multifunctional ability of the computer lies in its adaptability to perform a set of instructions repeatedly. This involves repeating some portion of the program either a specified number of times or until a particular condition is being satisfied. This repetitive operation is done through a loop control instruction. During looping, a set of statements are executed until some conditions for termination of the loop is encountered .A program loop consists of two segments, one is the **body of the loop** and the other known as the **control statement**. The control is tested always for execution of body of the loop.

Depending on the position of the control statement in the loop, a control may be classified as the **entry controlled loop** or as the **exit controlled one** (Fig.8.1). In the **entry** controlled loop, the control condition is tested first and if satisfied then only body of the loop is executed. In the **exit controlled** loop, the test is made at the end of the body, so the body is executed unconditionally first time.



(a) Entry controlled loop



(b) Exit controlled loop

Fig.8.1 loop control s

A looping process, in general, would include the following four steps:

1. Setting and initialization of a counter.
2. Execution of the statement in the loop
3. Test for a specified condition for execution of the loop.
4. Incrementing the counter.

The three loop constructs in C language for performing loop operations are:

1. The ***while*** statement
2. The ***do-while*** statement
3. The ***for*** statement.

Sentinel loops

Based on the nature of control variable, and the type of value assigned to it, for testing the control expression, there are two types of loops:

1. counter controlled

2. sentinel controlled loops (repetition).

Counter controlled repetitions are the loops which the number of repetitions needed for the loop is known before the loop begins; these loops have control variables to count repetitions. Counter controlled repetitions need initialized control variable (loop counter), an increment (or decrement) statement and a condition used to terminate the loop (continuation condition).

Sentinel controlled repetitions are loops with an indefinite repetitions; this type of loop use a special value, called **sentinel** value, to change the loop control expression from true to false(i.e., to indicate end of iteration) .

2.2 The *While* statement.

While statement is a s**entinel c**ontrolled repetition which can be iterated infinite number of times. Number of iterations is controlled using the **sentinel** variable (test expression). It is one of the simplest looping structures. The basic format of the **while** statement is:

```
While (test condition)
  {
      body of the loop
   }
```

The ***while*** is an **entry-controlled** loop statement. The test condition is evaluated and only if the condition is true the body is executed. After execution of the body, the test-condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test-condition finally becomes false and the control is transferred out of the loop. On exit, the program continues with the statement immediately after the body of the loop. If the body contains only one statement it is not necessary to put the braces, but placing them is a good programming practice. Let us look at a simple example, which uses a ***while*** loop.

```
# include< stdio.h>
int main( )
 {
    int p,n,count;
    float r,si;
    count =1;
    while(count <= 4)
      {
            printf (”enter values for p,n,r\n”);
            scanf ( “%d %d %f “, &p,&n,&r”);
            si = p*n*r/100;
            printf(“Simple interest is: Rs. %\n f”, si);
            count = count +1;
        }
        return 0;
   }
```

Fig 8.2: program to illustrate ***whil***e loop

Here, the program executes all the statements after ***while*** 4 times. The logic for calculating the simple interest is written within a pair of braces (i.e., the statements form body of while loop) immediately after the keyword while. The parentheses after the while contain a condition. So long as this condition remains true, all statements within the body of the ***while*** loop keeps getting executed repeatedly. .Also, to start with, the variable **count** is initialized to 1 and every time the logic of simple interest is executed, the value of count is incremented by one .The index variable **coun**t here, is called the loop counter .

The following points about ***while*** are worth noting.

1. The statements within w***hile*** loop would keep on getting executed till the condition being tested remains true. When the condition becomes false, the control passes to the first statement that follows the body of the ***while*** loop.
2. In the place of condition there can be any other valid expression. So long as the expression evaluates to a non zero value, the statements within the loop would get executed.
3. The condition being tested may be relational or logical operators as in the example below.

```
while (i < = 4)
while (i > = 4 && j < = 5)
while (i >. = 4 && ( j < 5 || c< 10))
```

4. The statements within the loop may be a single line(i.e., here braces optional) or a block of

Statements as in example shown below.

```
               while( i < =5)
                 i = i+1;
is same as,    while( i < =5)
               {
                  i = i+1;
               }
```

5. Almost always, the ***while*** must test a condition that will eventually become false, otherwise the loop

Will be executed for ever.

6. Instead of incrementing a loop counter (not necessarily integer it can be a float), one can Decrement it and can still manage the body of the loop to be executed repeatedly.

2.3 The *Do* Statement

The ***do while*** loop is also a kind of loop, which is similar to the ***while*** loop, in contrast to while loop, the ***do while*** loop tests at the bottom of the loop after executing the body of the loop. Since the body of the loop is executed first and then the loop condition is checked we can be assured that the body of the loop is executed at least once. The ***while*** on the other hand, will not execute its statements if the condition fails for the first time. That is, the *w**hile*** tests the condition before executing any of the statements within the ***whil**e* loop. As against this, the ***do-while*** tests the condition after having executed the statements within the loop. Since the test condition is evaluated at the bottom of the loop, *the* ***do-while*** statement is

```
do
{
        body of the loop
}
while (test condition);
```

an **exit controlled** loop statement. The ***do-while*** loop looks like this: Here the statement is executed first, and next the expression is evaluated. If the condition in the expression is true then the body is executed again and this process continues till the conditional expression becomes false. When the expression becomes false the loop terminates. This difference is brought about more clearly by the following program.

```
#include<stdio.h>
int main ( )
 {
     while ( 4<1)
             printf("hello\n");
     return 0;
  }
```

Here the, since the condition fails the first time itself, the printf ( ) will not get executed at all. The same program using **the *do-while*** construct is

```
#include<stdio.h>
int main ( )
{
   do
    {
          printf("hello\n");
      } while ( 4<1);
      return 0
  }
```

In this program, the printf ( ) would be executed once, since first the body of the loop is executed and then the condition is tested. ***Break*** and ***continue*** are used with ***do while*** just as they would be in a ***while***. A ***break*** takes one out of the ***do-while*** by passing the conditional test. A ***continue*** sends you straight to the test at the end of the loop.

2.4 The For Statement

The for loop is another entry-controlled loop that provides a more concise loop control structure. It is a counter controlled repetition. Therefore the number of iterations **must** be known before the loop starts (or predetermined). The body of a **for** statement is executed zero or more times until an optional condition becomes false. Also one can use optional expressions with in the **for** statement to initialize and change values during the for statements execution. **Th**e general form of the **for** loop is:

```
for (initialization; test condition; increment;)

{

      body of the loop

}
```

That is, in the control block of the **for** loop statement there are three expressions separated by semicolon (;).The execution of the for loop is as :

1. **The initialization**: Initialization of the control variables is done first using assignment statements .It is typically used to initialize a loop counter variable.
2. The value of the control variable is tested using the **test condition**. The test condition is a relational expression, such as i <5 that determines when the loop will exit. That is, the loop condition expression is evaluated at the beginning of each iteration. The execution of the loop continues until the loop condition evaluates to false.
3. **Increment**: The increment expression is evaluated at the end of each iteration. It is used to increase or decrease the loop counter variable.

Let us write down the simple interest program(which we have written earlier using **while** statement) using **for (Fig.8.3).** If this program is compared with the one written using **while** construct, we can see that , the three steps of f**o**r loop construct have now been incorporated in the **for** statement. Here in this program (fig 8,3), when the **for** statement is executed for the first time, the value of c**ount** is set to an initial value 1. Next the condition count <=3 is tested. Since the count was set to 1, the condition is satisfied and the body of the loop is executed for the first time. Up On reaching the closing brace of for**, control** is sent back to the **for** statement, where the value of count is incremented by 1. Again the test is performed to check whether the new value of count exceeds 3. If the value of count is less than or equal to 3, the statements within braces of for are executed again,. The body of the for loop continues to get executed till count does not exceed the final value 3.The control exits from the loop , when count reaches the value 4.and the control is transferred to the statement(if any) immediately after the body of **for.**

```
#include<stdio.h>
int main( )
 {
        int p,n,si;
        float,si;
        for(count =1; count <=3; count= count+1)
         {
               printf(enter the values for p.n,r\n”);
               scanf(“%d %d %f”,&p,&n,&r);
               si = p*n*r/100;
               printf(“ simple interest + rs. %f\n”, si);
               return 0
          }
  }
```

.

Fig 8.3: Program using for loop

Additional Features of **for** loop

1. More than one variable can be initialized at a time in the **for** statement as in :

for (p =1, n =6; n <11; $^{++}$n)

Statement. That is, initialization section has two parts p = 1 and n = 6 , separated by comma..Like

initialization section, increment section too can have more than one part. The multiple arguments in

the increment section too are separated by commas.

2. The test condition may have any compound relation and the testing need not be limited only to the

Loop control variable. For eg:

```
sum = 0;
for ( i =1 ;i<10 && sum< 19; ++i )
{
  S = s+1;
 printf("%d %d \n",i,sum):
}
```

Here the loop uses a compound test condition with the counter variable i and variable sum .The loop is executed as long as both the conditions i<10 && sum < 19 are true. The sum is evaluated inside the loop.

3. It is also permissible to use expressions in the assignment statements of initialization and increment Sections. For eg. A statement of the type

```
for( x= (m + n)/2; x > 0; x = x/2)
```

is valid.

4. One or more sections can be omitted if necessary as in eg.,

```
 -------------------
 m=5;
for (; m ! = 100 ;)
 {
        printf( " %d\n", m);
        m = m+3;
 }
```

Here, both initialization and increment sections are omitted in the **for** statement. The initialization has been done before the **for** statement and the control variable is incremented inside the loop. Though the sections remains blank, the semicolons separating the sections must remain. If the test condition is not present, the **for** statement sets up an infinite loop. Such loops can be broken using **break or goto** statements in the loop..

5.Time delay loops in **for** loop can be set up using the null statement as:

```
for ( i = 100; i > 0; i = i-1)
;
```

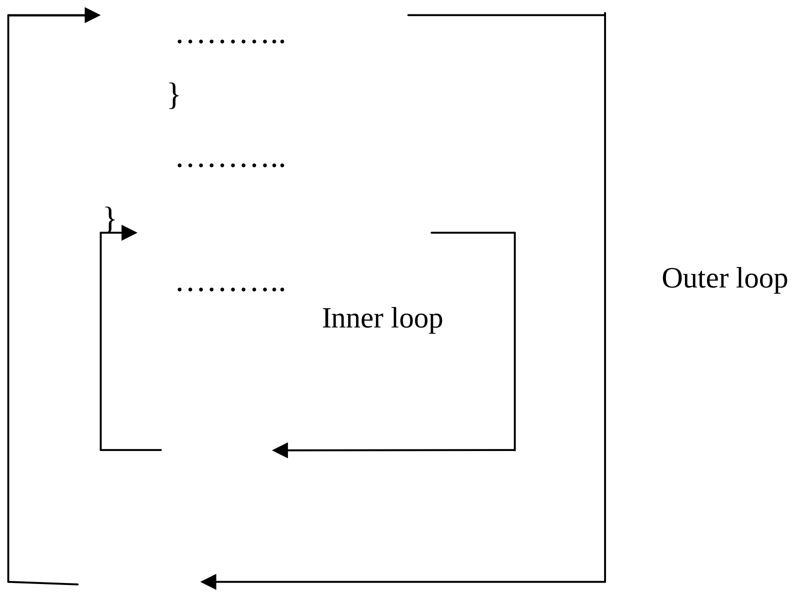
Here this loop is executed 100 times without any output. The body of the loop contains only a semicolon.

Known as null statement.

Nesting of For Loops

The way IF statements can be nested, similarly whiles and **for**s can also be nested; two loops can be nested as follows:

```
…………
…………..
for ( i =1; i <10; ++ i)
{
        …………
    for ( j= 1; j!  = 5: ++ j)
    {
        ………..
      }
        ………..
}
        ………..
```



The nesting may continue up to any desired level. To understand how nested loops work, we look at the program below.

```
# include< stdio.h>
int main ( )
 {
     int r,c,sum;
     for ( r =1; r < =3; r ++)
      {
            for( c=1; c<=2; c++)
             {
                  sum = r+c;
                  printf("r= %d sum = %d \n", r,c,sum);
               }
      }
        return 0;
 }
```

output

```
r =1 c=1 sum=2
 r =1 c=2 sum=3
r =2 c=1 sum=3
r =2 c=2 sum=4
r =3 c=1 sum=4
r =3 c=2 sum=5
```

Fig 8.4. Program to explain *neste*d **for**

Here for each value of r, the inner loop cycles through twice, with variable c taking values 1and 2.The inner loop terminates when c exceeds 2 and the outer loop terminates when r exceeds 3.

2.5 Jumps in loops

We often come across situations, where we want to jump out of a loop instantly, without waiting to get back to the conditional test. The keyword **break** allows to do this. When **break** is encountered in a loop , control automatically passes to the first statement after the loop. A **beak** is usually associated with an **if.** The key word **break, b**reaks the control only from the **while** in which it is placed. As an example we have :

```
# include < stdio.h>
int main( )
  {
     int num, i;
     printf(“ enter a number”);
     scanf(“%d”, & num);
     i =2;
     while( i < = num-1)
       {
         if (num% ! = = 0)
           {
               printf( “not a prime number\n”);
               break;
           }
           i++;
       }
       if ( i = = num)
              printf(“prime number\n”);
   }
```

Fig 8.5 use of **brea**k statement

2.6 The continue statement

The keyword **continu**e, allows us to take the control to the beginning of the loop, by passing the statements inside the loop, which have not yet been executed. That is , when the key word **continue** is encountered inside any loop, control automatically passes to the beginning of the loop .A **continue** is usually associated with an **if.** The **syntax** is:

Continue;

```
#include < stdio.h >

main()
{
    int i;
    int j = 10;

    for( i = 0; i <= j; i ++ )

    {

       if( i == 5 Goods 1

)
       {
          continue; Goods 1

       }
       printf("goods %d\n", i );
    }
}

Output

Goods 1

Goods 2

Goods 3

Goods 4

Goods 5

Goods 6

Goods 7

Goods 8

Goods 9

Goods 10
```

Fig .8. 6 .Use of **continue** statement

As an example consider the program of Fig.8.6. The use of **continue s**tatement in loops is illustrated in fig 8.7.In **while** and **do while** loops, **continue**, causes the control to go directly to the test condition and then to continue the iteration process. In the case of **for** loop, , the increment section of the loop is executed before the test condition is evaluated.

```
While (test condition)          do                          for(initialization; test condition; increment)
 {                               {                          {
   ………………..                      ………                         ……………..
   If (……………)                   if(………)                     if(…………..)
      Continue;                      continue;                   continue;
  ………………                          ………..                      ……………..
  ………………                          …………                       ……………..
   }                              } (while test condition );    }
```

Fig.8.7 *continue* **command in while, do while and for loop statements**

Jumping out o

We have seen that we can jump out of a loop using either the **break** or **goto** statement. In the same way we can jump out of a program by using the library function exit( ).. The use of exit( ) function is shown in fig. 8.8 below:

```
…………..
………….
 If (test condition) exit (0);
……………
……………
```

Fig.8.8. use of **exit** ( ) function.

2.7 Summary:

1.The three types of loops available in C are for, while, and do while.
2. A Break statement takes the execution control out of the loop.
3.a continue skips the execution of the statements after it and takes he control to the beginning of the loop.
4. A do while loop is used to ensure that the statements with in the loop are executed at least once.
5 when we need to choose one among number of alternatives, a switch statement is used.
6.The switch key word is followed by an integer or an expression that evaluates to an integer.
7. the case keyword is followed by an integer or a character constant.
8. the usage of goto keyword should be avoided as it usually violates the normal flow of execution.

Module IV: Introduction

This module is designed as an introduction to Data structures. It is about structuring and organizing data as a fundamental aspect of developing computer application. The standard data structures which are often used and which forms the basis for complex data structures is the array. An array is a homogenous data structure in which all elements are of the same type. In the first unit of the module, we describe different types of arrays in general. The next unit is devoted to a useful introduction to User defined functions.

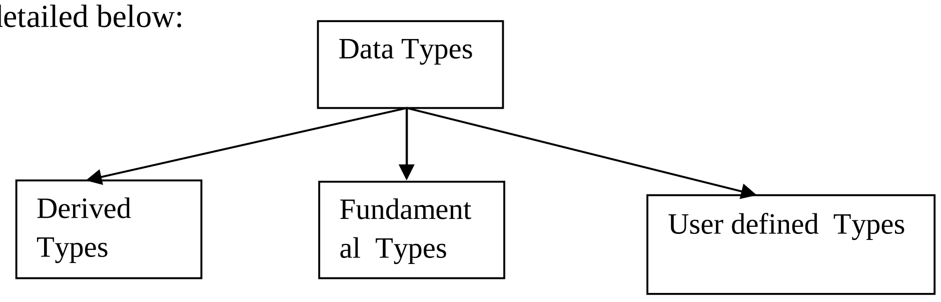
Unit 1 :Arrays

Structure

1.1 Introduction

An array is a collection of similar elements. These similar elements could be all integers, or all floats, , or all characters, etc. Usually, an array of characters is called a ‘ string’, where as an array of integers or floats is simply called an **array**. All elements of any given array must be of the same type. That is, we cannot have an array of 10 numbers, of which five are of integers and five of float type.

C supports a rich set of derived and user defined data types, in addition to a variety of fundamental data types.as detailed below:



Arrays	Integral types	Structures
Functions	Float types	Unions
Pointers	Character Types	Enumerations.

Arrays and structures are referred to as **structured data types** because they can be used to represent data values that have a structure of some sort. Structured data types provide an organizational scheme that shows the relationship among the individual elements and facilitate efficient data manipulation. In programming language such data types are known as **data structures**.

1.2 One dimensional Arrays.

As already discussed**,** an array is a collective name given to a group of similar variables .The values in an array is called as **elements** of array, and are accessed by numbers called **subscripts**. The array which is used to represent and store data in a linear form (or accessing its elements involve only a single subscript) is called as single or **one dimensional array**. As an example consider the C declaration:

```
int number [5];
```

Here in this declaration, the array variable **number** contain 5 elements of any value available to the i**nt** type .and the computer reserves 5 storage locations. The values to the array elements can be assigned as:

```
number [0]= 12;

number [1]=13;

number [2]=15;

number [3]=20;

number [4]=25;
```

This would cause the array number to store the values as shown below:

number [0]	12
number [1]	13
number [2]	15
number [3]	20
number [4]	25

These elements may be used in programs just like any C variable

1.3 Declaration of one dimensional Arrays

To begin with, like other variables an array needs to be declared before they are used so that the compiler will know what kind of an array and how large an array we want. The general form of array declaration is:

type variable-name [size];

The **type** specifies the type of element that will be contained in the array, such as int, float or char and the **siz**e indicates the maximum number of elements that can be stored inside the array .For example,

```
int marks [10];
```

Declares the marks as an array to contain a maximum of 10 integer constants. This number is often called the **dimension** of the array .The bracket ([ ]) tells the compiler that we are dealing with an array.

The C treats character strings simply as array of characters. The size in a character string represents the maximum number of characters that the string can hold. For instance,

```
char name[13];
```

Declares the name as a character array(string) variable that can hold a maximum of 13 characters. Suppose we read the following string constant in to the string variable name

“GOOD MORNING”

In this, each character of the string is treated as an element of the array name and is stored in the memory as:

```
‘G’
‘O’
‘O’
‘D’
 ‘ ‘
‘M’
‘O’
‘R’
‘N’
‘I’
‘N’
‘G’
‘\o’
```

When the compiler sees a character string , it terminates with an additional null character \o. Thus the element name[13] holds the null character '\o'. Remember that, while declaring character arrays, we must allow one extra space for the null terminator.

1.4 Initialization of one dimensional Array.

After an array is declared, its elements must be initialized. If they are not given any specific value, they are supposed to contain garbage values. An array can be initialized at either of the following stages:

- at compile time
- at run time

Compile time initialization

Whenever we declare an array we can initialize it directly at compile time. In this type of initialization, we assign certain set of values to array elements before executing program The general form of initialization of arrays is:

```
type array-name[ size ] = [ list of values ];
```

the values in the list are separated by commas. The type size can be specified directly as :

```
int num [5] = { 2.3,4,5,6};
```

Here the size of the array is specified directly as 5 in the initialization statement. The compiler will assign values to the particular elements of the array. i.e., At the time of compilation all, the elements are at specified positions as shown below.

```
num [0] = 2
num [1] = 3
num [2] = 4
num [3] = 5
num [4] = 6
```

Also the type size can be specified indirectly as in:

```
int num [ ] = { 2.3,4,5,6};
```

The compiler counts the number of elements written with in the braces and determines the size of the array.

Character arrays may be initialized in the same manner. Thus the statement

```
char name [ ] = { 'j', 'o', 'h', 'n', '\o'};
```

Declares the name to be an array of five characters, initialized with the string 'john' ending with the null character. Alternatively, we can assign the string literal directly as :

```
char name [ ] = 'john';
```

Run time initialization

An array can also be explicitly initialized at run time usually; .this approach is applied for initialization of large arrays. For example, consider the following program segment;

```
for (i  = 0;  i < 5; i++)
{
 scanf ( “% d “’  & x [ i ] );
}
```

The above segment will initialize the array elements with the values entered through the keyword .In this type of initialization (run time initialization) of the arrays. looping elements are almost compulsory. Looping statements are used to initialize the values of the arrays one by one by using assignment operator or through the keyboard by the user. we can also use read function such as **scanf** to initialize an array as in example below.

```
int x [2] ;
```

```
    # include < stdio.h >
    void  main  ( )
     {
        int array [3], i;
        printf( “ enter 3 numbers to store them in an array\n” );
        for ( i =0; i < 3; i ++)
         {
            scanf ( “ % d “, & array [ i] ) ;
         }
         printf ( “ elements  in the array are: \ n”);
         for  i =0; i < 3; i ++)
          {
            printf (“ elements stored at a [ %d] = %d\n”,i, array [ i]);
          }
          getch ( );
   }
```

output

```
enter  3 elements in the array : 2 3 4
elememts in the array are :
element stored at a[ 0] = 2
element stored at a[ 1] = 3
element stored at a[ 2] = 4
```

Fig 9.1: program to illustrate an array

```
scanf ( " %d % d",  & x[0], & x[1] );
```

will initialize the array elements with the values entered through the key word. Here is a sample program (Fig.9.1) to store the elements in the array and to print them from this array.

Searching and sorting are two operations performed on arrays. Searching is the process of arranging elements in the list according to their values, in ascending or descending order. An ordered list is a sorted one. The three simple and important sorting methods are:

Bubble sort

Selection sort

Insertion sort.

Other sorting methods include, Merge sort, quick sort and Shell sort.

Searching is the process of finding the location of the specified element in a list. The specified element is often called the **search key**. If the process of searching finds a match of the search key with a list element value, then the search is sad to be successful. Otherwise it is unsuccessful. Two most commonly used searching methods are ;

Sequential search

Binary Search.

1.5 Two dimensional Arrays.

So far, we have explored arrays with only one dimension. It is also possible to have two or more dimensions. The 2-D array is also called a matrix. The 2-D arrays are declared as :

```
type array-name  [ size   of row] [ column size ];
```

2-D arrays are stored in memory as shown below. In memory, whether, it is single or two dimensional array, the array elements are stored in one continuous chain .Each dimension of the array is indexed from zero to its maximum size minus one: the first index selects the row and the second index selects the column within that row,

Column 0

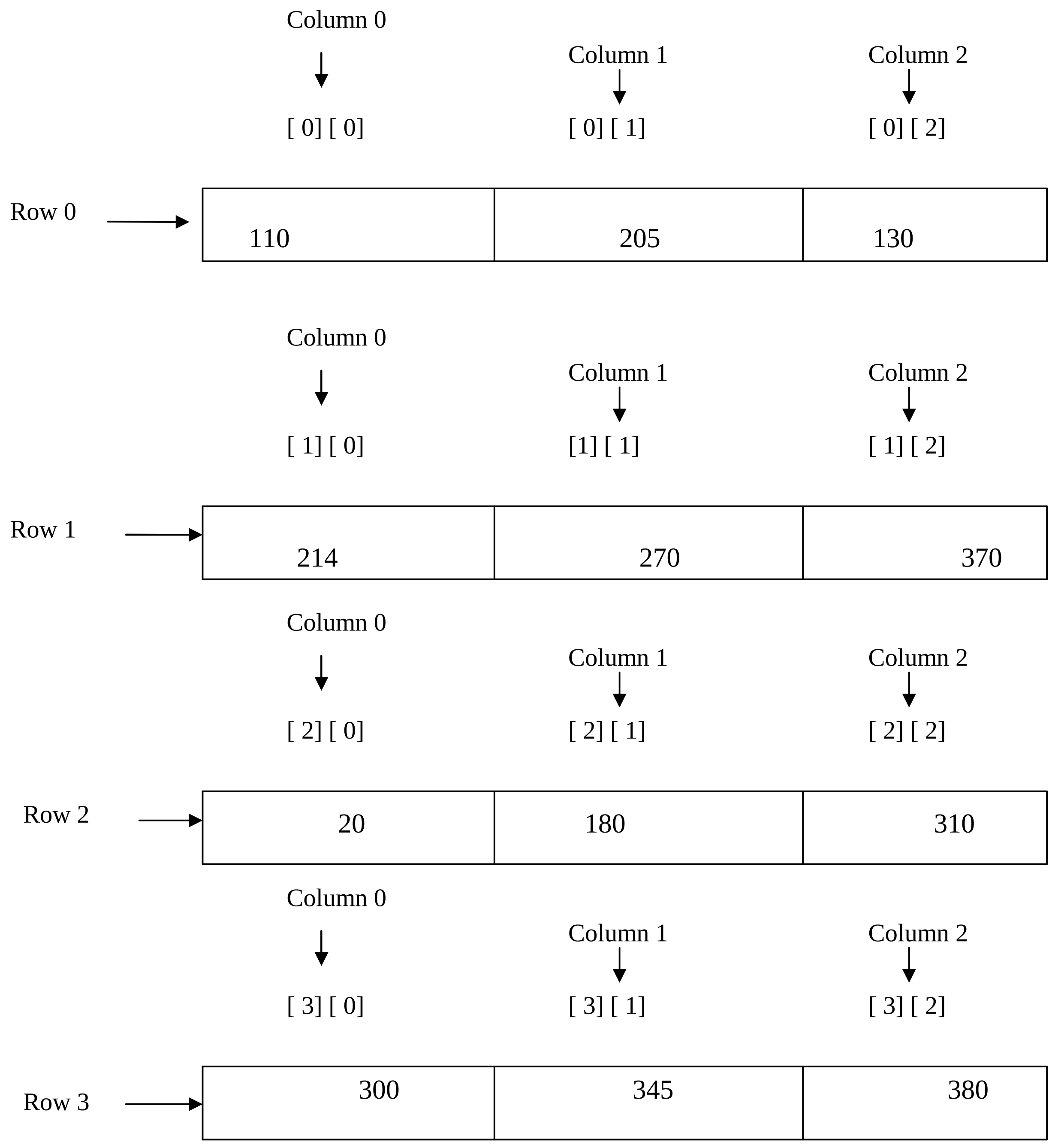


Fig 9,2 : Representation of 2-D array in memory

Here is a sample program:

```
# include< stdio.h>
int main( )
{
    int students [4] [2];
    int i,j;
    for (i = 0; i < = 3; i ++)
     {
          printf ( “enter the roll no of student and marks\n”);
          scanf(“ %d %d”,  &student [i] [0], &student[i][1]);
       }
    for (i =0; i< = 3; i ++)
           printf( “ %d %d “, student [i] [0],student [i] [1]);
    return 0;
  }
```

9.3. program to illustrate 2-D array

This program stores the roll number and marks obtained by a student side by side in a matrix. In the first part of the program, i.e., in the first f**or** loop, we read in the values of roll number and marks, where as in the second **for** loop, we print out these values. Also, in the first **scanf** , the first subscript of the variable student is row number which changes for every student. The second subscript tells which of the two columns are we talking about- the zeroth column which contains the roll number or the first column which contains the mark. The counting of rows and columns begins with zero. Remember that two dimensional array is a collection of a number of one dimensional arrays placed one below the other .In this program, the array elements have been stored row wise and accessed row wise. Although it is possible to access the elements column wise, row-wise strategy is accepted widely.

1.6 Initializing 2-D arrays

Like 1-D arrays, 2-D arrays could be initialized by following their declaration with a list of initial values enclosed in braces as in ,

```
int table  [2][3] = { 0,0,0,1,1,1};
```

which initializes the first row to zero and second row to one. Equivalently one can write the above statement as:

```
int table  [2][3] = {{ 0,0,0} ,{ 1,1,1}};
```

We can also initialize a 2-D array in matrix form as:

```
int table  [2][3] = {
                      {0,0,0},
                      {1,1,1}
                     };
```

More over, the declaration

```
int table  [ ][3] =  {
                      { 0,0,0},
                      {1,1,1}
                     };
```

Is perfectly valid.

If the values are missing in the initatializer, they are automatically set to zero. For instance, the statement

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

will initialize the first two elements of the first row to one, the first element of the second row to 2 and all other elements to zero.

In situations where we have to initialize all the elements to zero, a short cut method as in,

```
int m [3] [5] = { { 0}, { 0},{0} };
```

may be used. Here the first element of each row is explicitly initialized to zero, while all other elements are automatically initialized to zero. the following statement would also work.

```
int m [ 3] [5] =- { 0,0};
```

1.7 Multi dimensional Arrays

The general form of a multidimensional Array is:

```
Type array-name [ s1] [s2] [s3] …….[sm] ;
```

Where si is the size of the ith dimension. A 3-D array can be thought of as an array of arrays of array. The outer array has three elements, each of which is 2-d array of four 1-D arrays., each of which contains two integers. That is, a 1-D array of two elements is constructed first, followed by placing four 1-D arrays placed one below the other. So that a 2-d array containing four rows is obtained. Thereafter, three 2-D arrays are placed one behind the other to yield a 3-D array containing three 2-D arrays.

1.8 Dynamic Arrays

In C it is possible to allocate memory to arrays at run time. The arrays created at run time are called dynamic arrays .Dynamic arrays are created using memory management functions like malloc, calloc, realloc, that are included in the header file< stdlib.h > The concept of dynamic arrays is used in creating and manipulating data structures like lists, stack and queues.

1.9 Summary:

1,An array is similar to an ordinary variable except that it can store multiple elements of similar type.

2.The array variable acts as a pointer to the zeroth element of the array. In 1-D array, zeroth element is a

single valued one, whereas in a 2-D array this element is a 1-D array. During multidimensional

initialization, omission of array size other than the first dimension may result an error.

3. While initializing character array, enough space is to be provided for the terminating null character.

4. The subscript variables in a array need to be initialized before they are used.

Unit 2:User Defined Functions

Structure

2.1 Introduction

The **C** language is similar to most modern programming languages in that, it allows the use of **functions** (i.e., a self contained block or module of program code), to get its tasks done. In general, **C** functions contain a set of instructions enclosed by braces'{ }' , that can perform a coherent task of same kind. They are easy to define and are reusable. That means, it can be executed from as many different points ***in a C program as required. Broadly speaking, the two categories of functions in C are (1) Library*** functions and (2) ***user defined*** functions. **Library functions** are in built functions that are grouped and placed together in a common place called 'library', and are capable of performing specific operations. The main difference between a **library** and **user defined function** is that **library** functions are not required to be written by the user where as a **user defined function** has to be developed by us at the time of writing a program. In fact, a **user defined function** later becomes a part of the C program library. **main** is a specially recognized function in C and **is a**n example of **user defined function** while the functions **printf a**nd **scanf** belong to the category of **library** function.

2.2 Need for User defined Functions

A **function in C,** is a module of a program code (or a block of code that takes information in, does some computation, and returns a new piece of information based on the parameter information) which deals with a particular task. In fact, every program can be thought of as a collection of these functions. That is, **functions** groups a number of program statements into a unit and this unit can be invoked from other parts of a program. This division approach clearly results in a number of advantages:

1.It results in high level modular programming, (Fig.10.1) wherein the high level logic of the overall problem is solved first while the details of each lower level functions are addressed later.

2. By using functions at the appropriate places, the length of the source program can be reduced.

3. A function may be used by many other programs.

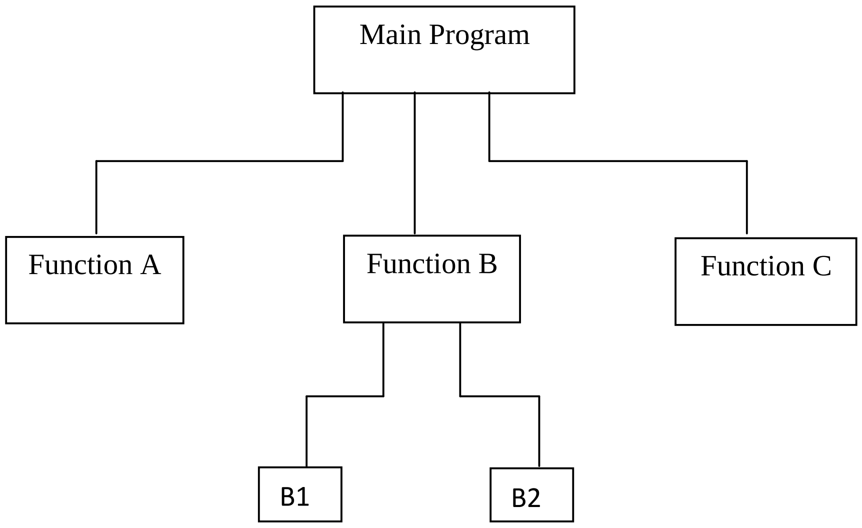


Fig.10.1 Top down Modular Programming using **functions**

2.3 A Multi function Program

As was pointed earlier, a **function** is a self contained block of instructions that perform a coherent task of some kind. Moreover, a **function** can be accessed from any location with in the C program. Making **functions** is a way of isolating one block of code from other Independent blocks of code. A **function** can take a number of parameters, do required processing and then return a value. When a function is

```
void message( );
int main ( );
{
   message( );
   printf ( “this explains  the use\n”);
   return 0;
 }
void message( )
 {
  printf ( “this is function definition \n”);
}
```

Fig 10.2

Defined at any place in the program then it is called function definition. That means, once a function is defined and packed, then it takes some data from the main program and returns a value. Actually, we will be looking at two things - a function that calls the function and the function itself. Let us consider the above chunk of program(fig.10.2).

And here is the output…..

```
this is function definition
this explains  the use
```

Here we have defined two user defined functions- **main ( )** and **message ( ).** In fact, we have used the word message at three places in the program. During the execution of the main, the first statement encountered is

message( );

which indicates that the function **message** is to be executed. At this point , the program transfers its control to the function **message**. After executing the **message** function (here no value is returned as was indicated by the key word **void**)., the control is transferred back to the **main**. Now, the execution continues at the point where the function call (by definition) was executed. After executing the **printf** statement, the control is again transferred to the function **message ( )** if being called by **main ( ).** That means the activity of **main ( )** is temporarily suspended while the **message ( )** function wakes up and goes to work. When the message ( ) function runs out of statements to execute, the control returns to **main ( ),** which comes to be active again by executing its code at the exact point where it left off. Thus, **main ( )** becomes the *calling* and message ( ) becomes the *called* function.

Any function can call any other function, In fact, it can call itself. Further, a called function can call another function. Also, a function can be called more than once in any program. Moreover, there are no predetermined relationships, rules of precedence or hierarchies (except at the starting point), among the functions that make up the complete program. The functions can be placed in any order and the called function can be placed either before or after the calling function. The best practice is to put all the called functions at the end. Figure 10.3 illustrates the flow of control in a multifunction program

2.4 Elements of User defined Functions

So far we have discussed and used a variety of data types and variables in our programs .Nevertheless, declaration and use of these variables were primarily done inside the main function. We can therefore define functions and use them like any other variables in C program. Both functions names and variables are considered as identifiers and therefore they must follow the rules for identifiers..Further, Like variables, functions have type associated with them and the function names and types must be declared and defined before they are used in program. Every user defined functions has three elements.

- Function definition
- Function Call
- Function Declaration.

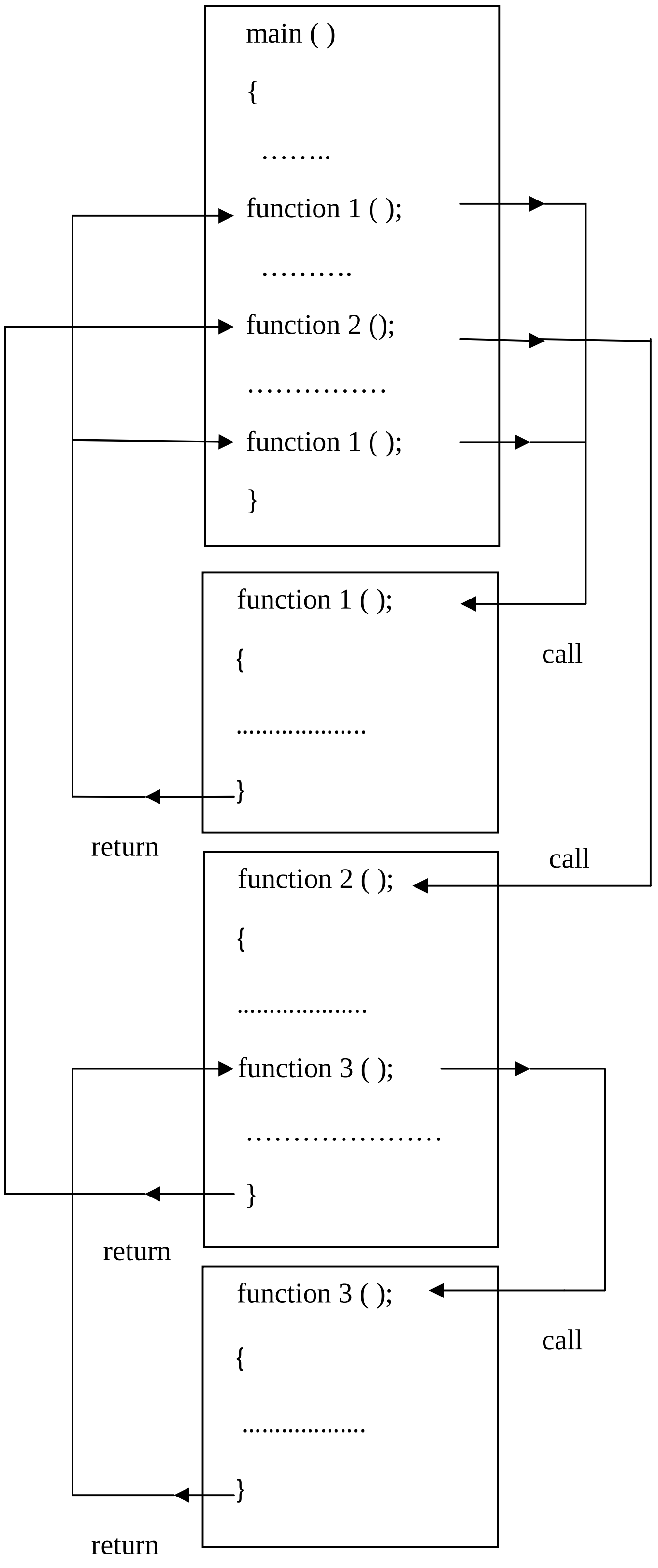


Fig 10.3 Flow of control in a multifunction program

The ***function definition*** is independent program modules that is specially written or apply the requirements of the function. To use this block or function, we need to call down it at the required place in the program, known as the functions. A function is defined when function name is followed by a pair of braces in which one or more statements may be present. The program that calls the function is referred to as the calling program or calling functions. The calling program should declare any function that is used later in the program. This is termed the function declaration or ***function prototype.***

2..5 Definition of Functions

The function ***definition*** which is the heart of function, is an independent program module that is specially written to suit to the requirements of the function. A function definition shall include the following elements

- Function name
- Function type
- List of parameters.
- Local variable declarations
- Function statements
- A return statement

All the six elements are grouped in two parts namely,

1. Function header (first three elements)

2. Function body (Second three elements)

A general format of function definition to implement these two parts(Fig.10.4) is:

```
function_ type  function_name (parameter list)

{

   local variable declaration;

   executable statement1;

   executable statement2;

   …………….

   …………….

   return statement;

 }
```

The first line **function_ type function_name (parameter list)** is known as the function header and the statements within the opening and closing braces constitute the function body.

Function header

The function header consists of three parts: function type, function name and the function parameter list. Semicolon is not used at the end of the function header.

Function name and type

Function type may specify the data type that one may use (like ***float ,int or double*** whatever according to ones needs) .If data type is not specified then C will assume it as ***int*** type and if the function does not return any value then ***void*** is used.

Function name may consist of any variable that is suitable for users understanding. That means, it is any valid C identifier that must follow the same rules of formation as other variable names in C. A function gets called when the function name is followed by a semicolon.

Parameter List

It declares the variables that are to be used in the function and that are going to be called in the program. Actually, they serve as input data to the function to carry out the specified task and are also be used to send values to calling programs. They are often termed as ***formal*** parameters(or arguments). The parameter list contains declaration of variables separated by commas and enclosed in parentheses with no semicolon after the closing parentheses. Note that combined declaration of parameter variables is invalid. That is, *int sum(int a,b)* is not a valid declaration of parameter list. To indicate an empty parameter list, usually we use the key word ***void*** between the parentheses as in

```
void printline (void)
 {
    ………..
   }
```

Many compilers do accept an empty set of parentheses, without specifying anything as in

```
void printline ( )
```

Again, its nice to have *void* to indicate a nill parameter list.

Function Body

The function body contains the declarations and statements necessary for performing the required task. The bodies enclosed in braces contain three parts:

- Local declaration that specify the variables needed by the function
- Function statements that perform the task of the function
- A ***return*** statement that returns the value evaluated by the function.

If the called function is not going to return any meaningful value to the calling function, the use of ***return*** statement can be omitted. Nevertheless, its return type should be specified as ***void***. But it is better to have a return statement even for ***void*** functions.

2.6 Return Values and their types

As pointed out earlier, a ***return*** statement is a statement that returns the value evaluated by the function to the calling program. If a function does not return any value, one can omit the ***return*** statement. When a ***return*** is encountered, the control is immediately passed back to the calling function. A function can ***return*** only one value at a time per call and the ***return*** statement can take one of the following forms:

return;

or

return (expression) ;

Here, the first 'plain' return does not return any value (or it acts as the closing brace of function).The second form of return returns the value of the expression. For example, the function

```
int mul ( int x, int y)
 {
    int z;
    z = x* y;
    return (z);
  }
```

Returns the value of z . It is possible to have more than one ***return*** statement for a function as in:

```
if ( x < = 0 )
     return ( 0 );
else
     return (1 );
```

All functions by default return ***int*** type data. We can force a function to return a particular type of data by specifying the ***type specifier*** in the function header. For functions that use ***doubles***, yet returns ***ints***, the returned value will be truncated to an integer as in:.

```
int product (void)
 {
 return (2.5* 3.0);
}
```

Will return the value 7, only the integer part of the computation.

2.7 Function Calls .

In order to use functions user need to call on it at a required place in the program. This is known as the function call. A function can be called by simply using the function name followed by a list of actual parameters, if any, enclosed in parentheses. For example,

```
main ( )
 {
   int y;
   y = mul (10, 5);             /* function call * /
   printf ( " %d\n",y);
 }
```

Here in the **main( )** program the mul(10, 5) function has been called. The C compiler, when it encounters a function call, the control is transferred to the function mul ( ). This function is then executed line by line and a value is returned (which is assigned to y) , when a return statement is encountered.

A Function that returns value can be used in expressions like any other variable.

```
e.g;   y = mul ( p,q)/(p+q);
```

Of course, a function cannot be used on the RHS side of an assignment statement. Thus, the statement

```
mul ( a,b) = 15;
```

Is wrong. Moreover a function , that does not return any value may not be used in expressions; but can be used to perform certain tasks specified in the function. Such functions may be called in by simply stating their names as independent statements. For example,

```
main ( )
{
   printline ( );
}
```

2.8 Function Declaration

The program or a function that called a function is referred to as the calling function or calling program. The calling program should declare any function that is to be used later in the program. This is known as the function declaration (also known as function prototype). Like variables, all the C functions must be declared, before they are called on. A function declaration involves four parts. viz,

- Function type
- Function name
- Parameter list
- Terminating semicolon.

The general format is:

Function- type function –name (parameter list);

The format is similar to the function header line except the terminating semicolon. Further, when a function does not take any parameters and does not return any value, its proto type , written as:

```
void  display (void);
```

A proto type declaration may be placed in two places in a program:

1. Above all functions including main (also called Global prototype);
2. Inside a function definition.(also called local prototype).

Global declarations are available for all the functions in the program where as local prototype type declarations are used by the functions containing them. The place of declaration of a function defines a region (also called *scope* of the function) in a program in which the function may be used by other functions. It is nice to declare prototypes in the global declaration section before ***main*** so that the user gets a quick reference to the functions used in the program thereby enhancing the documentation.

2.9 Category of Functions .

A function depending on whether arguments are present or not and whether a value is returned or not, may be categorized into:

- Functions with no arguments and no return values
- Functions with arguments and no return values
- Functions with arguments and one return values
- Functions with no arguments but return a value
- Functions that return multiple values

Let us have a look category of functions one by one.

2.10 Functions with no arguments and no return values

When a function has no arguments, the called function does not receive any data from the calling function and it does not return any data back to the calling function. Hence there is no data transfer between the called and calling function. This is pictorially represented in Fig. 10.5.Let us understand this with the help of a program (Fig 10.6)

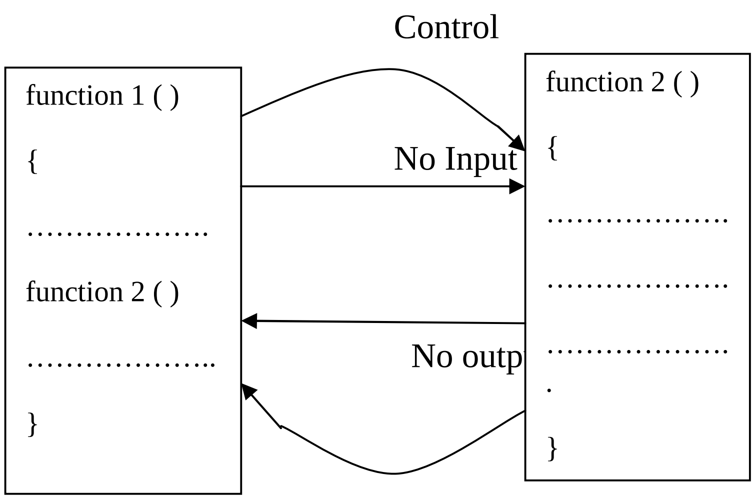


Fig 10.5

Control

```
void main ( )
{
 read_value ( );
}
read_value ( );
{
char name [10];
printf("enter your name\n");
scanf("%s", name);
printf("your name is % s, name");
}
```

output

```
enter your name
salu
your name is salu
```

Fig.10.6

2.11 Function with Arguments but no return value:

Here the called function receives the data from the calling function but the called function does not

```
# include < stdio.h>
# include < conio.h>
 void main ( )
{
 int a,b;
printf("enter the value for a and b\n");
scanf("%d %d", &a, &b );
largest (a,b);
largest (c,d);
int c,d;
{
if ( c > d)
{
printf(" largest = % d\n");
}
else
{
printf(" largest = % d\n");
}
return ( );
}
```

output

```
enter  the value for a and b
5
3
largest = 5
```

Fig 10.7   Program to find largest of two numbers

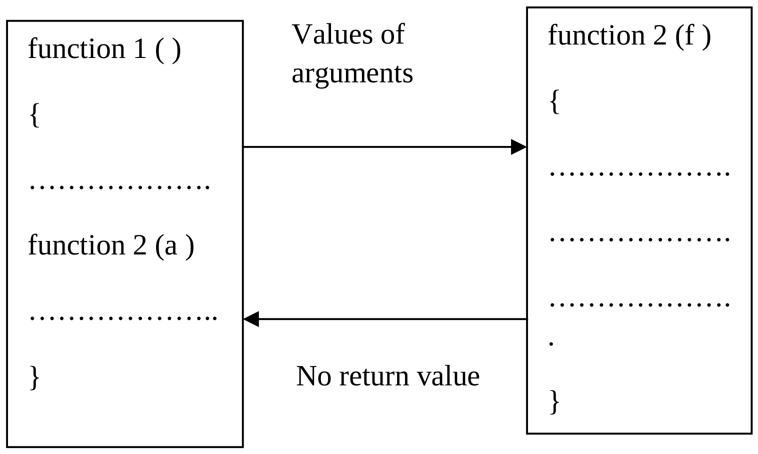


Fig 10.8

return any value back to the calling function. This is depicted in Fig 10.8.The dotted lines in Fig 10.8 Indicates that there is only transfer of control but not data.. A sample program to illustrate this is shown in Fig 10.7

2.12. Arguments with return values

In this type of functions, functions accepts arguments and returns value back to the calling program That means, a self contained and independent function receives a predetermined form of input and outputs a desired value. Thus it is a two way communication between a calling function and a called function (fig.10.9)

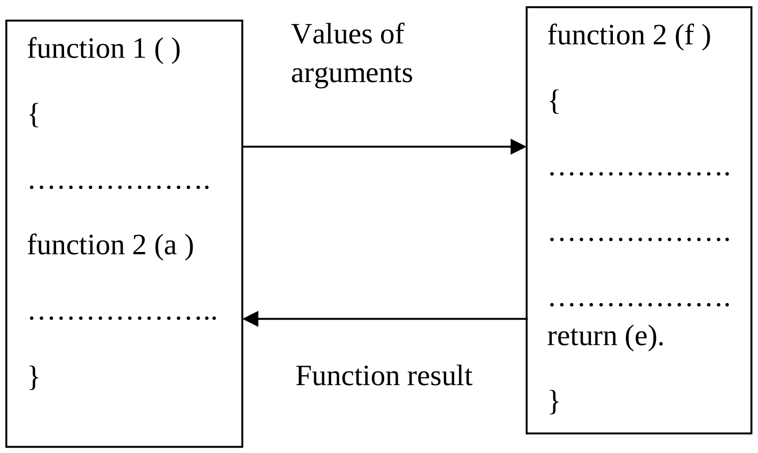


Fig 10.9

For example, the program (Fig.10.10) illustrates the use of two way data communication between calling and called functions.

```
#include < stdio.h >
float  calculate_ area ( int);
int main ( )
{
    int radius;
    int area;
    printf (“ enter the radius:\n”);
    scanf( “%d”, & radius);
    area = calculate_area(radius);
    printf(“ area of circle :”, area);
    return (0);
 }
float  calculate_ area ( int radius);
{
   float area of circle;
   area of circle = 3.14 * radius * radius;
   return(area of circle);
}
```

output

```
enter the radius: 1
area of circle = 3.14
```

Fig 10.10: program to show functions with argument and return value

2.13 Functions with no arguments but returns a value

In this type, the called function does not receive any data from the calling function. It is also a one way data communication between the calling function and the called function. To understand this following program (fig 10.11) will help.

```
# include < stdio.h >

# include < conio.h >

void main ( )

{

float sum;

float total ( );

clrscr ( );

sum = total ( );

printf ( " sum = % f \n", sum);

}

float total ( )

{

float a,b;

a = 2;

b= 8;

return (a+b);

}
```

output

```
sum = 10.000000
```

Fig 10.11: function with no arguments but returns a value

2.14 Functions that return multiple values

Using a return statement, a function in C can return only one value. If we want the function to return more than one value of same data types, we could return the pointer to array of that data types. We can also make the function return multiple values by using the arguments of the function. That is , by providing the pointers as arguments. In fact, when a function needs to return several values, we use one pointer in return instead of several pointers as arguments. Here, the mechanism of sending back information through arguments is achieved by using what are known as address operator (&) and indirection operator (*).For e.g., consider the program code:

```
void mathoperation ( int x, int y, int *s, int *d);
main ( )
{
    int x =10, y = 8, s,d;
     mathoperation (x, y, &s, &d);
     printf ( " s = % d \n d = % d\n", s,d);
}
void mathoperation 9 int a, in b, int * sum, int * diff )
 {
    *sum = a+b;
    *diff = a-b;
}
```

In this code, in the function call, when we pass the actual values of x and y to the function, we pass the address of locations where the values of s and d are stored in the memory. When a function call I s passed, the following assignments takes place.

Value of x to a

Value of y to b

Address of s to sum

Address of d to diff

The indirection operator *** (The name indirection means that it gives indirect reference to variable through its address)** in the declarations **sum** and **diff** in the header indicates these variables are to store addresses and not the actual values of variables. That means, the variables **sum** and **diff** point to the memory location of s and d respectively. In the body of the function, the statements

*sum = a + b;

* diff = a - b;

Imply that the value stored in the location pointed to by **sum** is the value of s and the value of a-b is stored in the location pointed to by **diff** is the value of d. The variables * sum and * diff are pointers and **sum** and **diff** are *pointer variables*..Since they are declared as i**nt** , they can point to locations of **int** type data. The use of pointer variables for communicating the data between functions is termed **call by reference (or call by address/ pass by pointers).**

2.15 Recursion

In C programming, it is possible for the functions to call themselves or the process of defining.

```
#include  < stdio.h>
int sum (int n);
int main ( )
{
     int num, add;
     printf( " enter a positive integer:\n");
     scanf(" %d", & num);
     add = sum (num);
     printf(" sum = %d ", add );
}
int sum(int n)
 {
    if (n = = 0 )
         return n ;
    else
        return n+ sum (n-1);
  }
output
enter a positive number
3
6
```

Fig.10.12 program code for the sum on n natural numbers

Something in terms of itself is known as recursion. A very simple example to find the sum of n natural numbers using recursion(or call a function inside the same function) is shown in Fig 10.12.In this example, the function sum ( ) is invoked from the same function. If n is not zero then the function calls itself by passing argument 1 less when the previous argument it was called with. When n becomes equal to zero, the value of n is returned .In this example, a better visualization of recursion for n = 3, assumes the form:

sum (3)

= 3+ sum (2)

= 3+2+sum(1)

= 3+2+1+sum(0)

= 3+2+1+0

= 3+2+1

= 3+3

= 6

That is, every recursive function must be accommodated with a way to end the recursion. when n is zero, there is no recursive function call and the recursion ends here.

2.16 passing arrays to function

In C programming it is possible to pass a single array or an entire array to a function. Also, both one and multidimensional array can be passed to function as argument. To pass a 1-d array to a called function, listing the name of the array without any subscripts, and size of the array as argument is sufficient. That means, while passing arrays to the argument, the name of the array is passed as an argument. Also, Single element of an array can be passed in the same way as passing variables to a function. For example, the following code

```
#include < stdio.h>
void display( int a)
    {
     printf(“%d”,a);
    }
 int main( ) {
    int c [ ] = {2,3,4};
    display ( c[2]);  /* passing array element c[2] */
    return 0;
}
```

Explains the passing single element of an array (**that is c[2])** to a function. The output of this program is 4.In C, the name of the array represents the address of its first element. By passing the array name in fact deals with passing the address of the array to the called function. The array in the called function refers to the same array stored in the memory. That is, any changes in the array in the called function will be reflected in the original array. Remember that one cannot pass a whole array by value, as we do in the case of ordinary variables. Also, when we deal with array arguments, care should be taken to incorporate the changes made to the original array that passed to the function, if the function changes the values of the elements of an array.

Two dimensional arrays

Like simple arrays, to pass two dimensional array to a function as an argument, the starting address of memory area reserved is passed .An example, to pass 2-D arrays to function is shown below.

```
# include < stdio.h>
void  function(int  c[2][2]);
int  main(){
int  c[2][2],i,j;
    printf("enter 4 numbers:\n");
    for(i=0;i<2;++i)
        for(j=0;j<2;++j){

scanf("%d",&c[i][j]);
        }
    function(c);
    return  0;
}
void  function(int  c[2][2]){
    int  i,j;
    printf("displaying:\n");
    for(i=0;i<2;++i)
        for(j=0;j<2;++j)
```

The output of this program is:

```
        Enter 4 numbers
      1
      2
      3
      4
    Displaying
     1
     2
     3
     4
```

The function defined in the program can be used in the main function to display 4 numbers in the array .

2.17. Passing strings to functions

The strings are treated as character arrays in C and therefore the rules for passing strings to functions are same as those for passing arrays to functions .The rules are as follows:

1. The strings to be passed must be declared as a formal argument of the function when it is defined.
2. The function prototype must show that the argument is a string. eg., **void display (char str [ ]);**
3. A call to the function must have a string array name without subscripts as its actual argument.

   eg. **display (names);**

2.18 Summary

1. Function declaration specifies the return type of the function and the types of parameters it accepts. A function can return only one value at a time.
2. There is no restriction on the number of return statements that may be present in a function. Also return statements need not always be present at the end of the called function.
3. A return statement is needed if the return type is anything other than **void.** If a function does not return any value, return type must be declared as **void**

4. Any number of arguments can be passed to a function being called. However, the type, order, and the number of actual and formal arguments must be same .If the value of the formal argument is changed in the called function, the corresponding change does not take place in the calling function.

5. Where more functions are used, they may be placed in any order.

6. If a function has no parameters, the parameter list must be declared as **void** .Functions return integer value by default.

7. Functions cannot be defined as assignment.

8. A function with void return type cannot be used in the RHS of an assignment statement.

9. Function definition defines the body of the function .it may be placed either after or before the main function.

10. Variables declared in a function are not available to other functions in a program.

11. A function can be called either by value or by reference.

12. Recursion offers a better solution than loops.

13. If a function is to be made to return more than one value at a time, then return these values indirectly by using a call by reference.

10 Use parameter passing by values as far as possible.