UNIT I

1

INTRODUCTION TO IMPERATIVE PROGRAMMING

Unit Structure

1.0 OBJECTIVES

Definition of Imperative Programming:

- The imperative (or procedural) paradigm is the closest to the structure of actual computers.

- It is a model that is based on moving bits around and changing machine state.

- Imperative programming is a programming paradigm that uses statements that change a program's state from compilation to running.

- In imperative language natural languages can be used to express commands for the computer to perform. Imperative programming focuses on describing how a program operates.

- In contrast to it, Declarative programming, which focuses on what the program should accomplish without specifying how the program should achieve the result.

Programming Languages based on the Imperative Paradigm have the Following Characteristics:

- The basic unit of abstraction is the PROCEDURE, whose basic structure is a sequence of statements that are executed in succession, abstracting the way that the program counter is incremented, so as to proceed through a series of machine instructions residing in sequential hardware memory cells.

- Variables play a key role, and serve as abstractions of hardware memory cells. Typically, a given variable may assume many different

values of the course of the execution of a program, just as a hardware memory cell may contain many different values. Thus, the assignment statement is a very important and frequently used statement.

- The sequential flow of execution can be modified by conditional and looping statements (as well as by the very low-level goto statement found in many imperative languages), which abstract the conditional and unconditional branch instructions found in the underlying machine instruction set.

1.1 INTRODUCTION

Imperative Programming can be Different Types of:
- Machine and Assembly languages i.e. Low level imperative language.
- Procedural languages i.e. High level imperative language.
- Structural & modular languages
- Object oriented languages
- Event Driven programming languages
- Object Based Languages

- Machine and Assembly Languages are native languages of a computer for hardware implementation it is designed and written in imperative style to execute in native code.

- Procedural Programming is a type of imperative programming in which the program is built from one or more procedures (also termed subroutines or functions). Procedural programming couldbe considered a step towards declarative programming. A programmer can often tell, simply by looking at the names, arguments, and return types of procedures (and related comments), what a particular procedure is supposed to do, without necessarily looking at the details of how it achieves its result. At the same time, a complete program is still imperative since it fixes the statements to be executed and their order of execution to a large extent.

- Structured Programming is a programming with a specific structure of the program and modular programming is added with structured language to add different functions. These are high level imperative languages with assignment statements, calculative statements, evaluation statements to execute complex expressions which may have arithmetic, relational & logical operators and function evaluations, and the assignment of the resulting value to memory. Looping statements like while, do while, for loop, etc. used to execute sequence, conditional branching, switch case statements and looping statements and subroutine or procedure call. Imperative languages are like Fortran, BASIC, Pascal, COBOL, ALGOL language for mathematical algorithms and C language,

- Object Oriented Programming are imperative in style, but added features to support objects. C++ , JAVA, Perl, Rubey, visual c++ are object oriented languages & Python languages

- Event Driven Programming languages are imperative style with object based events handlers Like Visual Basic & PHP with Web designating languages.

- Object Based Languages are programming languages with imperative style by introducing pure object oriented concepts and object based concepts by introducing VB.Net & C#, J# & F# functional languages.

- C is the Mother of all imperative types of programming languages, since from c language BASIC language is invented and from BASIC language Visual Basic & VB.Net languages are invented. From C language C++, VC++, C#, J#, JAVA languages are invented.

- Hence In this Book we are Considering C as a imperative Language & all examples are covered considering C language only.

Introduction to Programming Languages:

- Programming Language is a language used to communicate with the computer by writing programs.

- Programming language is widely used in the development of operating systems.

- An OperatingSystem (OS) is a software (collection of programs) that controls the various functions of a computer.

- Also it makes other programs on your computer work. For example, you cannot work with a word processor program, such as Microsoft Word, if there is no operating system installed on your computer.

- Windows, Unix, Linux, Solaris, and Mac OS are some of the popular operating systems.

- The same way to run the programs in a particular programming language we need a language complier.

- You write computer instructions in a computer programming language such as Visual Basic, C#, C++, or Java. Just as some people speak English and others speak Japanese, programmers write programs in different languages.

- The instructions you write using a programming language are called program code; when you write instructions, you are coding the program. Every programming language has rules governing its word usage and punctuation.

- These rules are called the language's syntax. Mistakes in a language's usage are syntax errors. After a computer program is typed using programming language statements and stored in memory, itmust be

3

translated to machine language that represents the millions of on/off circuits within the computer.

- Your programming language statements are called source code, and the translated machine language statements are object code.

- Each programming language uses a piece of software, called a compiler or an interpreter, to translate your source code into machine language.

- Machine language is also called binary language, and is represented as a series of 0s and 1s.

- The compiler or interpreter that translates your code tells you if any programming language component has been used incorrectly. Syntax errors are relatively easy to locate and correct because your compiler or interpreter highlights them. If you write a computer program using a language such as C++ but spell one of its words incorrectly or reverse the proper order of two words, the software lets you know that it found a mistake by displaying an error message as soon as you try to translate the program.

- After a program's source code is successfully translated to machine language, the computer can carry out the program instructions.

- When instructions are carried out, a program runs, or executes. some input will be accepted, some processing will occur, and results will be the output.

Types of Programming Languages:
1. Machine Level language
2. Assembly language
3. Procedural language (High Level Language)

Machine Language:

Every computer has its own language called machine language. It depends on the specific Hardware of the computer. A machine language is also known as low level language also called machine understandable language. Computer understands & executes the program only in machine level language. This low level language is in the form of (1's and 0's) binary codeLow Level Language requires memorizing or looking up numerical codes for every instruction that is used. These are machine dependent languages. These are used for simulation languages, and LISP, artificial intelligence applications.

Assembly Language:

Assembly language is the mnemonic language written in some specific symbolic codes, such as ADD, SUB etc. An assembly language program is first translated into machine language instruction by system program called assembler, before it can be executed. These Are languages

understandable by CPU & ALU section of Computer Assembly languages are called low level languages.

High Level Language:

A High level language is a simple English like language. A High levellay program also needs to be transferred into machine language instructions before it can be executed because computer understands only machine level language. Rules for programming in a particular high-level language are much the same for all computers, so that a program written for one computer can generally be run on many different computers with little or no alteration. This translation, called compilation is done by a systems program called a compiler. The original program written in High level language is called source program and its translation ie., machine code is called object program. Some popular High Level languages are Basic, Fortran, Cobol, Pascal, C & C++.High-level language offers three significant advantages over machine language: simplicity, uniformity and portability (i.e., machine independence).Compilers and Interpreters A program written in a high level language must be translated into machine language before it can be executed. This is known as compilation or interpretation, depending on how it is carried out.

Compilers translate the entire program into machine language before executing any of the instructions. Interpreters, on the other hand, proceed through a program by translating and then executing single instructions, or small groups of instructions. A compiler or interpreter is itself a computer program that accepts a high-level program (e.g. a C program) as input data, and generates a corresponding machine – language program as output. The original high-level program is called the source program, and the resulting machine-language program is called the object program. Every high-level language must have its own compiler or interpreter for a particular platform. It is generally more convenient to develop a new program using an interpreter rather than a compiler. Once an error-free program has been developed, a compiled version will normally be executed much faster than an interpreted version. Difference between compiler and interpreter

Sr.No	Compiler	Interpreter
1.	Compiler translates entire program into machine language at a time	Interpreter translates and interpretes line by line into machine language.
2.	It takes a large amount of time to analyze the source code but the overall execution time is comparatively faster	It takes less amount of time to analyze the source code but the overall execution time is slower.
4.	It generates the error message only after scanning the whole program. Hence debugging is	Continues translating the program until the first error is met, in which case it

	comparatively hard.	stops. Hence debugging is easy.
5.	Errors are displayed after entire program is checked and Intermediate Object Code is Generated	Errors are displayed for every instruction interpreted (if any) No Intermediate Object Code is Generated
6	Programming language like C, C++ use compilers.	Programming language like Python, Ruby use interpreters.

History of C Programming Language:

- **History of C language** is interesting to know. Here we are going to discuss a brief history of the c language.

- C programming language was developed in 1972 by Dennis Ritchie at bell laboratories of AT&T (American Telephone & Telegraph), located in the U.S.A.

- Dennis Ritchie is known as the founder of the c language.

- It was developed to overcome the problems of previous languages such as B, BCPL, etc.

- Initially, C language was developed to be used in UNIX operating system. It inherits many features of previous languages such as B and BCPL.

Let's see the programming languages that were developed before C language.

Language	Year	Developed By
Algol	1960	International Group
BCPL	1967	Martin Richard
B	1970	Ken Thompson
Traditional C	1972	Dennis Ritchie
K & R C	1978	Kernighan & Dennis Ritchie
ANSI C	1989	ANSI Committee
ANSI/ISO C	1990	ISO Committee
C99	1999	Standardization Committee

C language has evolved from three different structured language ALGOL, BCPL and B Language. It uses many concepts from these languages and has introduced many new concepts such as data types, struct, pointer. In 1988, the language was formalized by American National Standard Institute (ANSI). In 1990, a version of C language was approved by the International Standard Organization (ISO) and that version of C is also referred to as C89.

**Why the Name "C" was given to Language?:**
1. Many of C's principles and ideas were derived from the earlier language B. (Ken Thompson was the developer of B Language.) 2. BCPL

and CPL are the earlier ancestors of B Language 3. CPL is common Programming Language. In 1967, BCPL Language (Basic CPL) was created as a scaled down version of CPL 4. As many of the features were derived from —B‖ Language that's why it was named as —C‖. 5. After 7-8 years C++ came into existence which was first example of object oriented programming.

Features of C Language:

C is the widely used language. It provides many **features** that are given below.

1. Simple
2. Machine Independent or Portable
3. Mid-level programming language
4. Structured programming language
5. Rich Library
6. Memory Management
7. Fast Speed
8. Pointers
9. Recursion
10. Extensible
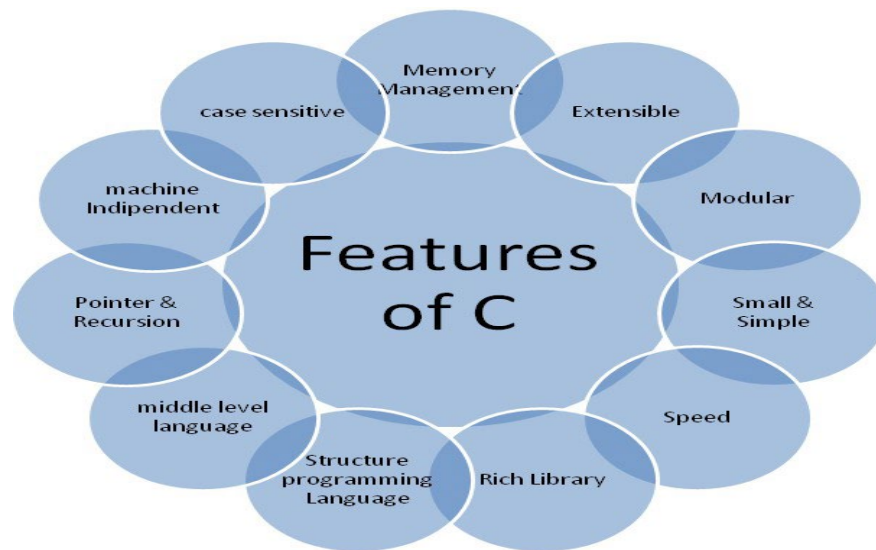
1) Simple:

C is a simple language in the sense that it provides a **structured approach** (to break the problem into parts), **the rich set of library functions**, **data types**, etc.

2) Machine Independent or Portable:

Unlike assembly language, c programs **can be executed on different machines** with some machine specific changes. Therefore, C is a machine independent language.

3) Mid-level programming language:

Although, C is **intended to do low-level programming**. It is used to develop system applications such as kernel, driver, etc. It **also supports the features of a high-level language**. That is why it is known as mid-level language.

4) Structured programming language:

C is a structured programming language in the sense that **we can break the program into parts using functions**. So, it is easy to understand and modify. Functions also provide code reusability.

5) Rich Library:

C **provides a lot of inbuilt functions** that make the development fast.

6) Memory Management:

It supports the feature of **dynamic memory allocation**. In C language, we can free the allocated memory at any time by calling the **free()** function.

7) Speed:

The compilation and execution time of C language is fast since there are lesser inbuilt functions and hence the lesser overhead.

8) Pointer:

C provides the feature of pointers. We can directly interact with the memory by using the pointers. We **can use pointers for memory, structures, functions, array**, etc.

9) Recursion|:

In C, we can call the function within the function. It provides code reusability for every function. Recursion enables us to use the approach of backtracking.

10) Extensible:

C language is extensible because it **can easily adopt new features**.

1.2 APPLICATIONS OF C PROGRAMMING

C was initially used for system development work, particularly the programs that make-up the operating system. C was adopted as a system

development language because it produces code that runs nearly as fast as the code written in assembly language. Some examples of the use of C are

- Operating Systems
- Language Compilers
- Assemblers
- Text Editors
- Print Spoolers
- Network Drivers
- Modern Programs
- Databases
- Language Interpreters
- Utilities

First C Program:

Before starting the abcd of C language, you need to learn how to write, compile and run the first c program.

To write the first c program, open the C console and write the following code:
1. #include <stdio.h>

2. **int** main(){

3. printf("Hello C Language");

4. **return** 0;

5. }

#include <stdio.h> includes the **standard input output** library functions. The printf() function is defined in stdio.h .

int main() The **main() function is the entry point of every program** in c language.

printf() The printf() function is **used to print data** on the console.

return 0 The return 0 statement, returns execution status to the OS. The 0 value is used for successful execution and 1 for unsuccessful execution.

How to compile and run the c program:
There are 2 ways to compile and run the c program, by menu and by shortcut.
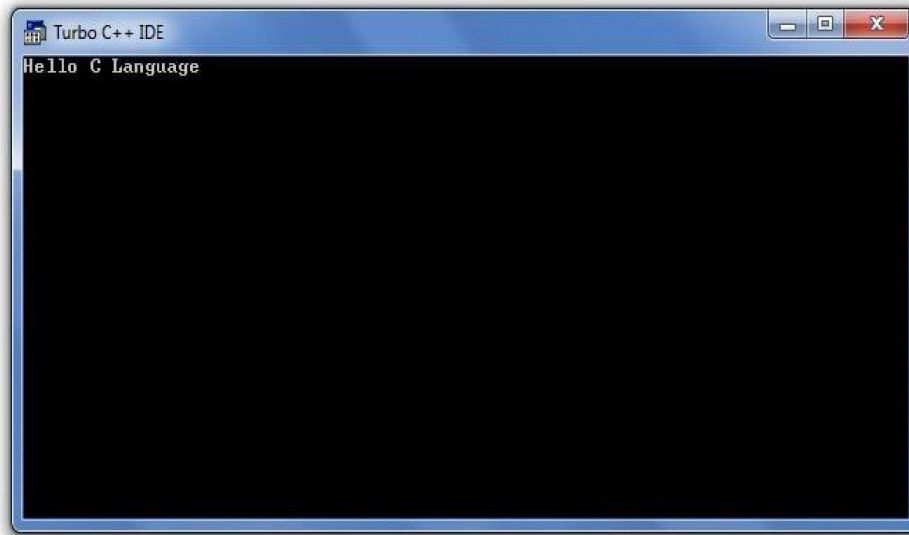
By menu:
Now **click on the compile menu then compile sub menu** to compile the c program.

Then **click on the run menu then run sub menu** to run the c program.

By shortcut

Or, press ctrl+f9 keys compile and run the program directly.

You will see the following output on user screen.

```
Turbo C++ IDE
Hello C Language
```

You can view the user screen any time by pressing the **alt+f5** keys

Now **press Esc** to return to the turbo c++ console.

1.3 PROGRAM DEVELOPMENT LIFE CYCLE

Program Development Cycle:

Programmer for developing the program can not directly start any program or project. Programmer has to understand whole program or project, has to analyze the sequence and flow of the program
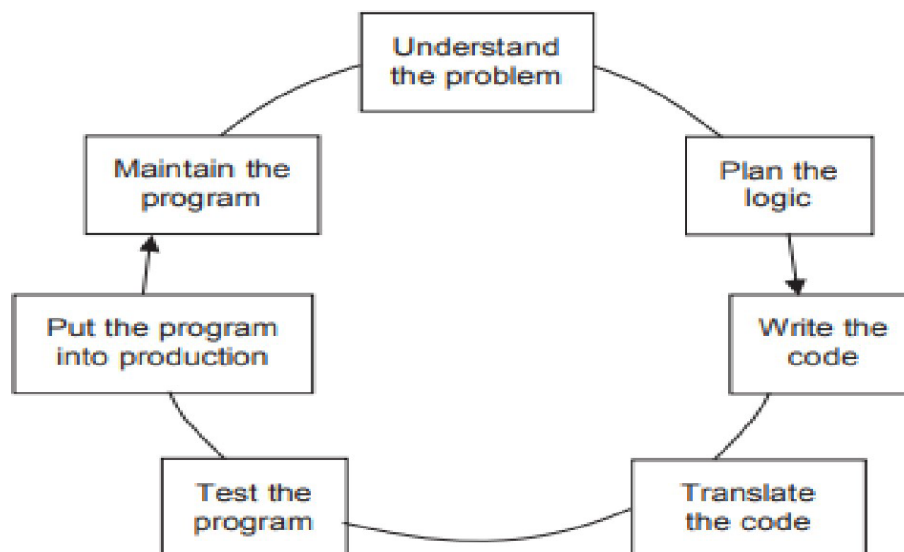
1. Understand the problem.
2. Plan the logic.
3. Code the program.
4. Use software (a compiler or interpreter) to translate the program into machine language.
5. Test the program.
6. Put the program into production.
7. Maintain the program.

1. Understanding the Problem:

Professional computer programmers write programs to accomplish the requirements of users or end users.

Examples of end users include payroll management system, they needs a printed list of all employees, a Billing department that wants a list of clients who are 30 or more days overdue on their payments, they need their deduction information. Since programmers are providing a service to these users, programmers must first understand what the users want. When a program runs, you usually think of the logic as a cycle of input processing-output operations, but when you plan a program, you think of the output first. After you understand what the desired result is, you can plan the input and processing steps to achieve it.

Suppose the manager needs a list of all employees who have been here over five years, because we want to invite them to a special thank-you dinner. On the surface, this seems like a simple request. An experienced programmer, however, will know that the request is incomplete. For example, you might not know the answers to the following questions about which employees to include: Does the manager want a list of full-time employees only, or a list of full and part-time employees together?

Does she want to include people who have worked for the company on a month-to month contractual basis over the past five years, or only regular, permanent employees?

Do the listed employees need to have worked for the organization for five years as of today, as of the date of the dinner, or as of some other cutoff date?

What about an employee who worked three years, took a two-year leave of absence, and has been back for three years?

The programmer cannot make any of these decisions; the user must address these questions to manager.

For example, no one knew they wanted to play Angry Birds or leave messages on Facebook before those applications were developed. Mobile app developers also must consider a wider variety of user skills than programmers who develop applications that are used internally in a corporation. Mobile app developers must make sure their programs work with a range of screen sizes and hardware specifications because software competition is intense and the hardware changes quickly.

2. Planning the Logic:

The main important part of the program is planning the program's logic. During this phase of the process, the programmer plans the steps of the program, deciding what steps to include. You can plan the solution to a problem in many ways. The two most common planning tools used are flowcharts and pseudocode. You may hear programmers refer to planning a program as ''developing an algorithm.'' An algorithm is the sequence of steps or rules you follow to solve a problem.

The programmer shouldn't worry about the syntax of any particular language during the planning stage, but should focus on figuring out what sequence of events will lead from the available input to the desired output. Planning the logic includes thinking carefully about all the possible data values a program might encounter and how you want the program to handle each scenario. The process of walking through a program's logic on paper before you actually write the program is called desk-checking.

3. Coding the Program:

After the logic is developed, only then can the programmer write the source code for a Program in a respective programming language. The logic developed to solve a programming problem can be executed using any number of languages. Only after choosing a language must the programmer be concerned with correct syntax.

4. Using Software to Translate the Program into Machine Language:

Even though there are many programming languages, each computer knows only one language— its machine language, which consists of 1s and 0s. Computers understand machine language because they are made up of thousands of tiny electrical switches, each of which can be set in either the on or off state, which is represented by a 1 or 0, respectively.

Languages like Java or Visual Basic are available for programmers because someone has written a translator program (a compiler or interpreter) that changes the programmer's English-like high-level programming language into the low-level machine language that the computer understands.

When you learn the syntax of a programming language, the commands work on any machine on which the language software has been installed. However, your commands then are translated to machine language, which differs in various computer makes and models.
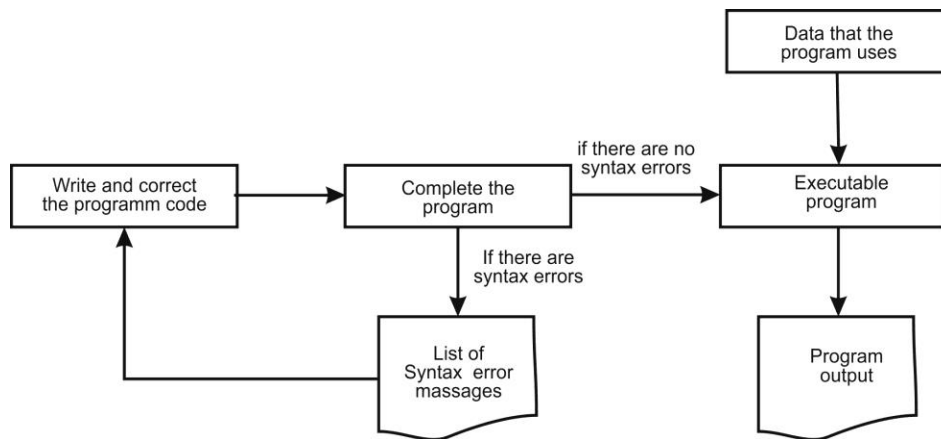

Diagram referred from programming logic and design by Joyce Farrell.

If you write a programming statement incorrectly the translator program doesn't know how to proceed and issues an error message identifying a syntax error.

Typically, a programmer develops logic, writes the code, and compiles the program, receiving a list of syntax errors. The programmer then corrects the syntax errors and compiles the program again. Correcting the first set of errors frequently reveals new errors that originally were not apparent to the compiler.

5. Testing the Program:

A program that is free of syntax errors is not necessarily free of logical errors. A logical error results when you use a syntactically correct statement but use the wrong one for the current context.

Input myNumber

setmyAnswer = myNumber * 2

outputmyAnswer

If you execute the program, provide the value 2 as input to the program, and the answer 4 is displayed, you have executed one successful test run of the program. Testing of logical errors, syntactically errors are done.

6. Putting the Program into Production:

Once the program is thoroughly tested and debugged, it is ready for the organization to use. Putting the program into production might

mean simply running the program once, if it was written to satisfy a user's request for a special list then we can finalize the program.

7. Maintaining the Program:

After programs is completed making necessary changes is called maintenance. Maintenance can be required for many reasons: for example, because new tax rates are legislated, the format of an input file is altered, or the end user requires additional information not included in the original output specifications. you make changes to existing programs, you repeat the development cycle. That is, you must understand the changes, then plan, code, translate, and test them before putting them into production.

1.4 USING PSEUDOCODE STATEMENTS AND FLOWCHART SYMBOLS

When programmers plan the logic for a solution to a programming problem, they often use one of two tools: pseudocode (pronounced ''sue-doe-code'') or flowcharts. Pseudocode is an English-like representation of the logical steps it takes to solve a problem. A flowchart is a pictorial representation of the same thing. Pseudo is a prefix that means ''false,'' and to code a program means to put it in a programming language; therefore, pseudocode simply means ''false code,'' or sentences that appear to have been written in a computer programming language but do not necessarily follow all the syntax rules of any specific language.

Writing Pseudocode:

You have already seen examples of statements that represent pseudo code earlier in this chapter, and there is nothing mysterious about them. The following five statements constitute a pseudocode representation of a number-doubling problem:

start
input myNumber
set myAnswer = myNumber * 2
output myAnswer
stop

Using pseudocode involves writing down all the steps you will use in a program. Usually, programmers preface their pseudocode with a beginning statement like start and end it with a terminating statement like stop. The statements between start and stop look like English and are indented slightly so that start and stop stand out. Most programmers do not bother with punctuation such as period sat the end of pseudocode statements, although it would not be wrong to use them if you prefer that style. Similarly, there is no need to capitalize the first word in a sentence, although you might choose to do so. This book follows the conventions of

using lowercase letters for verbs that begin pseudocode statements and omitting periods at the end of statements. Pseudocode is fairly flexible because it is a planning tool, and not the final product. Therefore, for example, you might prefer any of the following:

- Instead of start and stop, some pseudocode developers would use the terms begin and end.
- Instead of writing input myNumber, some developers would write getmyNumber or read myNumber.
- Instead of writing set myAnswer = myNumber * 2, some developers would write calculate myAnswer = myNumber times 2 or computemyAnswer as myNumber doubled.
- Instead of writing output myAnswer, many pseudocode developers would write display myAnswer, print myAnswer, or write myAnswer.

The point is, the pseudocode statements are instructions to retrieve an original number from an input device and store it in memory where it can be used in a calculation, and then to get the calculated answer from memory and send it to an output device so a person can see it. When you eventually convert your pseudocode to a specific programming language, you do not have such flexibility because specific syntax will be required. For example, if you use the C# programming language and write the statement to output the answer, you will code the following: Console. Write (myAnswer);

The exact use of words, capitalization, and punctuation are important in the C# statement, but not in the pseudocode statement.

1.5 ALGORITHMS AND FLOWCHART

Algorithms:
1. A sequential solution of any program that written in human language, called algorithm.
2. Algorithm is first step of the solution process, after the analysis of problem, programmer write the algorithm of that problem.
3. Example of Algorithms:

Q. Write am algorithm to find out number is odd or even?
Ans.
step 1 : start
step 2 : input number
step 3 : rem=number mod 2
step 4 : if rem=0 then
print "number even"
else
print "number odd"
endif
step 5 : stop

Flowchart:

Defination: Graphical representation of any program is called flowchart.

Flowchart is a diagrammatic representation of sequence of logical steps of a program. Flowcharts use simple geometric shapes to depict processes and arrows to show relationships and process/data flow.

Flowchart Symbols:
Here is a chart for some of the common symbols used in drawing flowcharts.

Symbol	Symbol Name	Purpose
	Start/Stop	Used at the beginning and end of the algorithm to show start and end of the program.
	Process	Indicates processes like mathematical operations.
	Input/ Output	Used for denoting program inputs and outputs.
	Decision	Stands for decision statements in a program, where answer is usually Yes or No.
	Arrow	Shows relationships between different shapes.
	On-page Connector	Connects two or more parts of a flowchart, which are on the same page.
	Off-page Connector	Connects two parts of a flowchart which are spread over different pages.

Guidelines for Developing Flowcharts:

These are some points to keep in mind while developing a flowchart −

- Flowchart can have only one start and one stop symbol
- On-page connectors are referenced using numbers
- Off-page connectors are referenced using alphabets

- General flow of processes is top to bottom or left to right
- Arrows should not cross each other

Example Flowcharts:
Here is the flowchart for going to the market to purchase a pen.
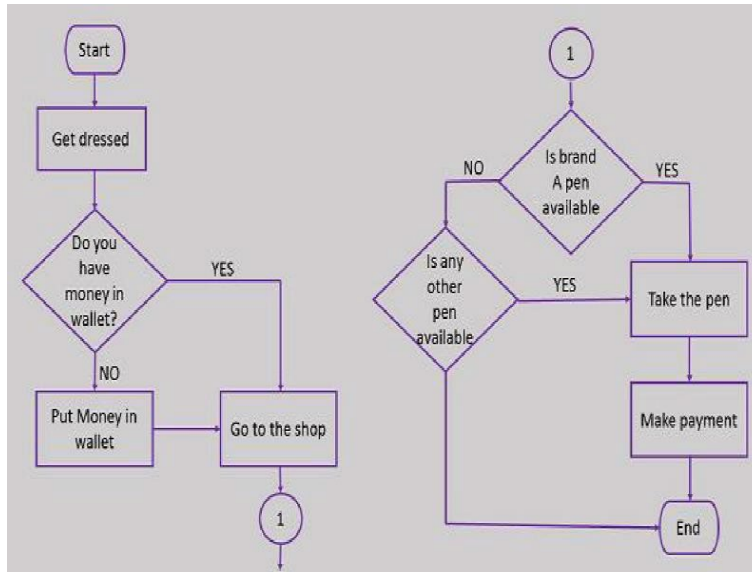
Here is a flowchart to calculate the average of two numbers.
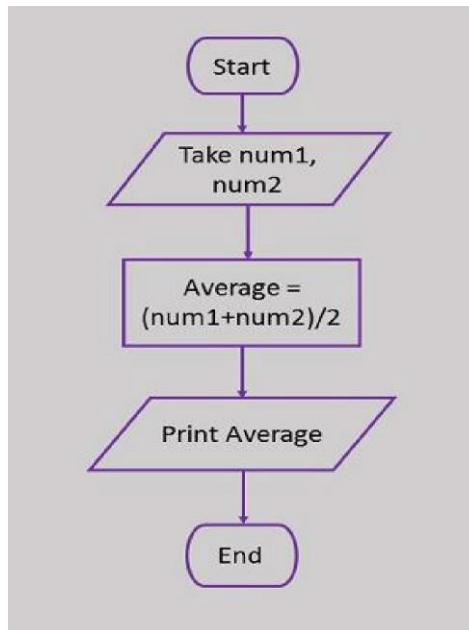
Sentinel Value to End a Program

Using a Sentinel Value to End a Program

The logic in the flowchart for doubling numbers, shown in Fig. 1.8, has a major flaw—the program contains an infinite loop.

17

If, for example, the input numbers are being entered at the keyboard, the program will keep accepting numbers and outputting their doubled values forever. Of course, the user could refuse to type any more numbers. But the program cannot progress any further while it is waiting for input; meanwhile, the program is occupying computer memory and tying up operating system resources.

Any infinite loop in a program has a problem that you can input values infinite time, processing will be done, result will be displayed. Loop will be continued infinite times. Solution to stop is turn off your computer, Solution is to somewhere Refuse input value or stop accepting input value.

A better way to end the program is to set a predetermined value for myNumber that means ''Stop the program!'' For example, the programmer and the user could agree that the user will never need to know the double of 0 (zero), so the user could enter a 0 to stop. The program could then test any incoming value contained in myNumber and, if it is a 0, stop the program.

Testing a value is also called making a decision.

You represent a decision in a flowchart by drawing a decision symbol, which is shaped like a diamond. The diamond usually contains a question, the answer to which is one of two mutually exclusive options— often yes or no.

The question to stop the doubling program should be ''Is the value of myNumber just entered equal to 0?'' or ''myNumber = 0?'' for short. The complete flowchart will now look like the one shown in Fig.
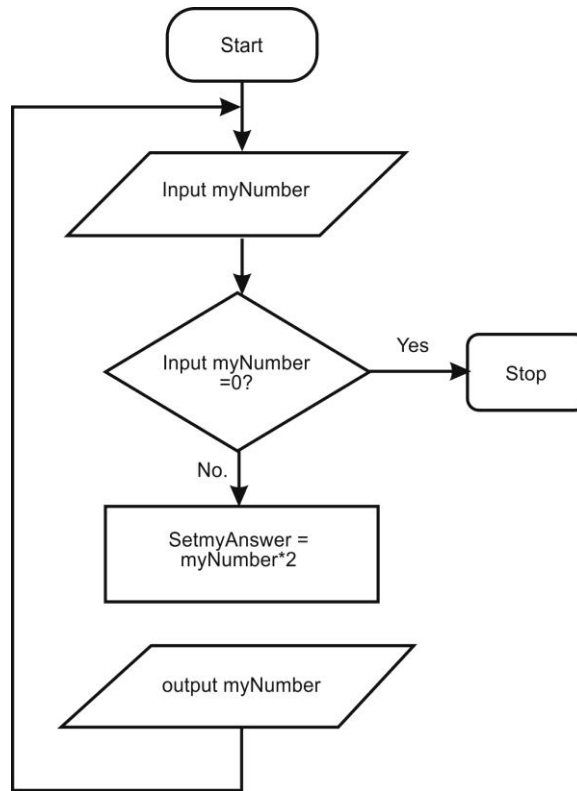
Flowchart with sentinel value equal to zero

One drawback to using 0 to stop a program, of course, is that it won't work if the user does need to find the double of 0. In that case, some other data-entry value that the user never will need, such as 999 or –1, could be selected to signal that the program should end.

A preselected value that stops the execution of a program is often called a dummy value because it does not represent real data, but just a signal to stop. Sometimes, such a value is called a sentinel value because it represents an entry or exit point, like a sentinel who guards a fortress.

For one thing, an input record might have hundreds of fields, and if you store a dummy record in every file, you are wasting a large quantity of storage on ''non data.'' Additionally, it is often difficult to choose sentinel values for fields in a company's data files.

Any balance Due, even a zero or negative number, can be a legitimate value, and any customerName, even ''ZZ'', could be someone's name. Fortunately, programming languages can recognize the end of data in a file automatically, through a code that is stored at the end of the data. Many programming languages use the terme of (for end of file) to refer to this marker that automatically acts as a sentinel.

Here In this example, therefore, uses eof to indicate the end of data whenever using a dummy value is impractical or inconvenient. In the flowchart shown in Fig. 1.10, the eof question is shaded.

19
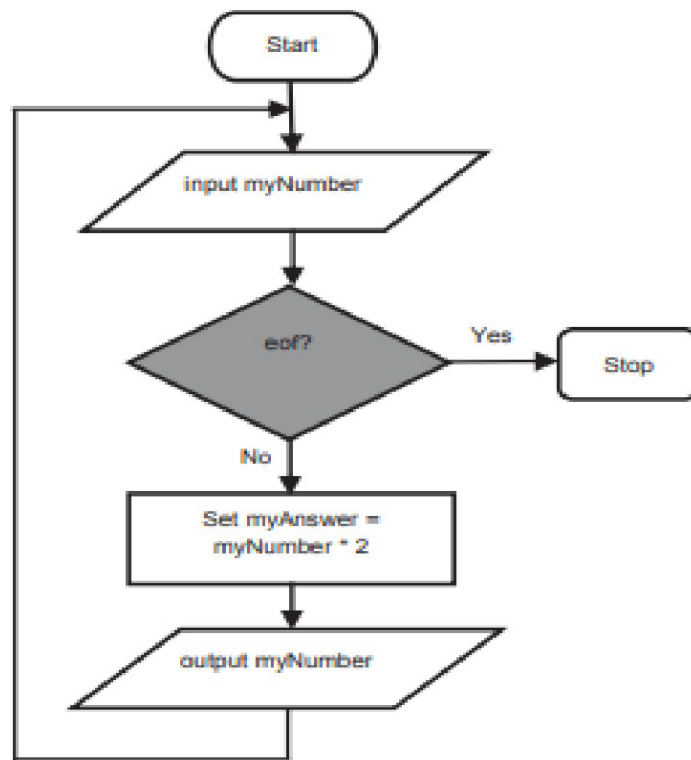
1.6 UNIT END QUESTIONS

1. What is Imperative Programming? What are its Types?

2. Write a Difference between Compiler and Interpreter.

3. Explain History of C Programming Language.

4. Enlist Features of C Programming. Explain any 4 in Brief.

5. Explain Program Development Life Cycle in Detail.

6. What is Flowchart ? What is the use of Flowchart?

*****

PROGRAMMING AND USER ENVIRONMENTS UNDERSTANDING PROGRAMMING ENVIRONMENTS

Unit Structure

2.0   Objectives

2.1   Introduction

2.2   Understanding the evolution of programming models:
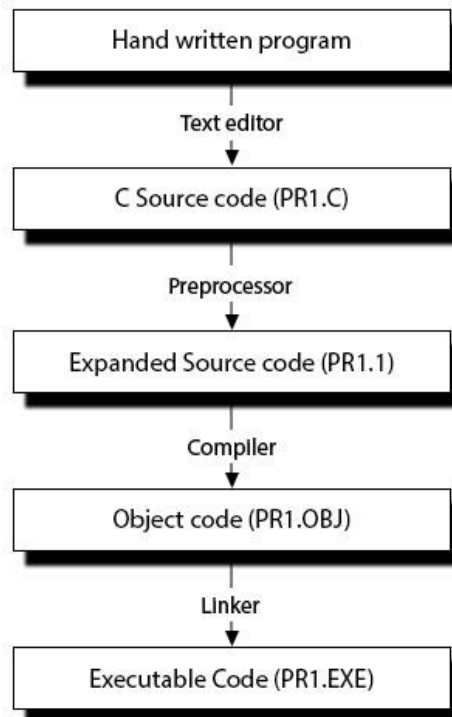
2.3   Unit End Questions

2.0 OBJECTIVES

Understanding Programming Environments

You can type a program into one of the following:

● A plain text editor
● Turbo c editor

A text editor that is part of an integrated development environment

A text editor is a program that you use to create simple text files. It is similar to a word processor, but without as many features.

You can use a text editor such as Notepad that is included with Microsoft Windows. The C Developing Environment is a screen display with windows and pull-down menus. The program listing, error messages and other information are displayed in separate windows.

The menus may be used to invoke all the operations necessary to develop the program, including editing, compiling, linking, and debugging and program execution.

If the menu bar is inactive, it may be invoked by pressing the [F10] function key. To select different menu, move the highlight left or right with cursor (arrow) keys. You can also revoke the selection by pressing the key combination for the specific menu.
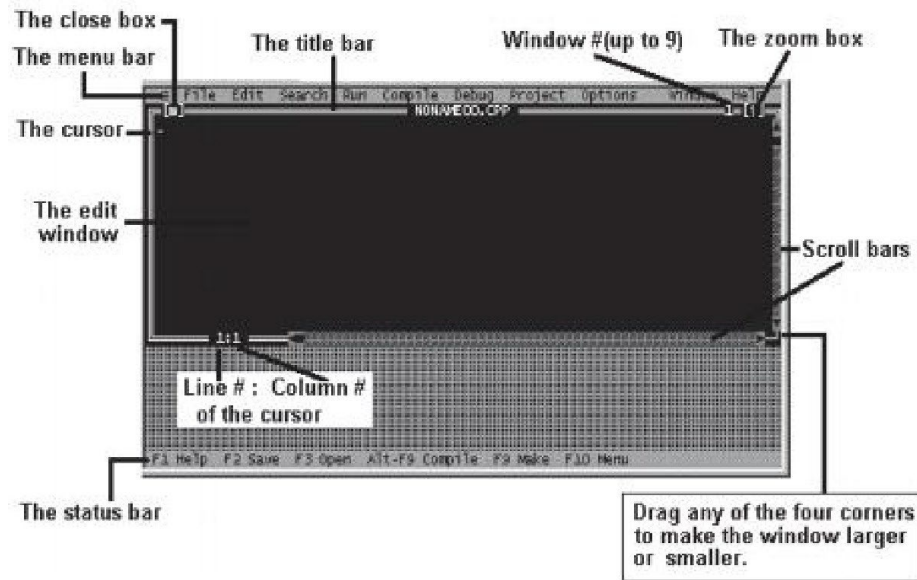
2.1 INTRODUCTION

Invoking the Turbo C IDE:

The default directory of Turbo C compiler is c:\tc\bin. So to invoke the IDE from the windows you need to double click the TC icon in the directory c:\tc\bin.

The alternate approach is that we can make a shortcut of tc.exe on the desktop. Opening New Window in Turbo C

To type a program, you need to open an Edit Window. For this, open file menu and click ''new''.
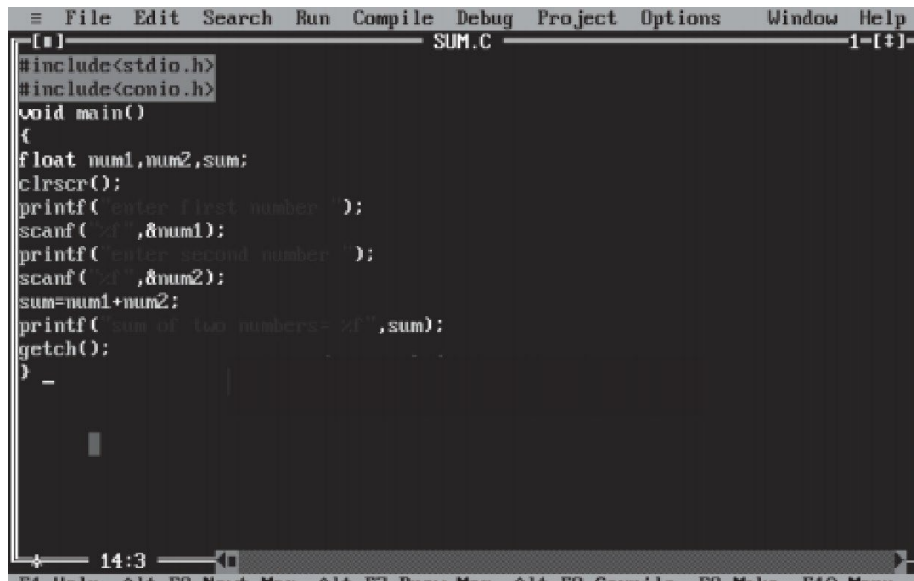
A window will appear on the screen where the program may be typed.

Writing a Program in Turbo C:

When the Edit window is active, the program may be typed. Use the certain key combinations to perform specific edit functions.

Saving a Program in Turbo C:

To save the program, select save command from the file menu. This function can also be performed by pressing the [F2] button. A dialog box will appear asking for the path and name of the file. Provide an appropriate and unique file name. You can save the program after compiling too but saving it before compilation is more appropriate

Making an Executable File in Turbo C:

The source file is required to be turned into an executable file. This is called ''Making'' of the .exe file. The steps required to create an executable file are:

1. Create a source file, with a .c extension.
2. Compile the source code into a file with the .obj extension.
3. Link your .obj file with any needed libraries to produce an executable program

All the above steps can be done by using Run option from the menu bar or using key combination

Ctrl+F9 (By this linking & compiling is done in one step).

Understanding User Environments:

Compiling and linking in the Turbo C IDE: In the Turbo C IDE, compiling and linking can be performed together in one step. There are

two ways to do this: you can select Make EXE from the compile menu, or you can press the [F9] key

Correcting Errors in Turbo C:

If the compiler recognizes some error, it will let you know through the Compiler window. You'll see that the number of errors is not listed as 0, and the word —Error‖ appears instead of the word —Success‖ at the bottom of the window. The errors are to be removed by returning to the edit window.

Usually these errors are a result of a typing mistake. The compiler will not only tell you what you did wrong, they'll point you to the exact place in your code where you made the mistake.

Executing a Programs in Turbo C:

If the program is compiled and linked without errors, the program is executed by selecting Run from the Run Menu or by pressing the [Ctrl+F9] key combination.

```
Enter the first number : 10 ∍
Enter the second number : 20
The sum of two numbers is: 30
```

Exiting Turbo C IDE An Edit window may be closed in a number of different ways. You can click on the small square in the upper left corner, you can select close from the window menu, or you can press the Alt+F3 combination. To exit from the IDE, select Exit from the File Menu or press Alt+X Combination.

Evolution of Programming Models:

Machine Language

Assembly Language

Procedure-oriented Language

Structured Programming Language

Object Oriented Programming
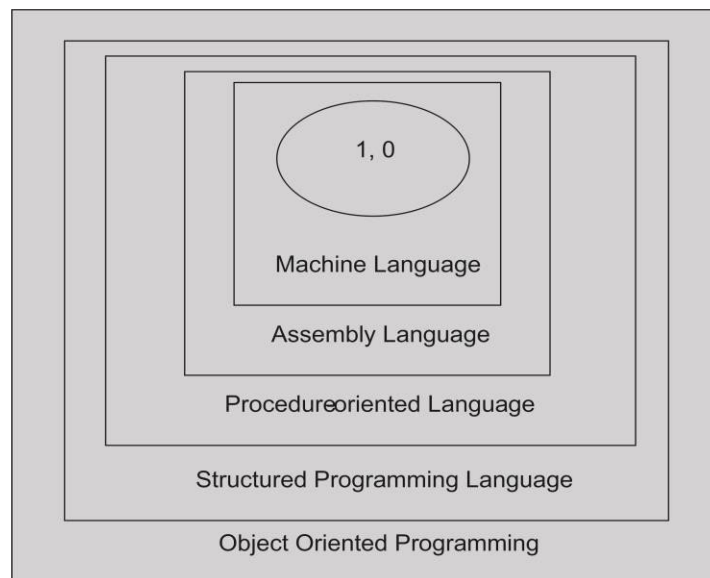
Software technology has a growth of a tree. Software evolution has a layer of growth. Each layer representing an improvement over the previous one.

The oldest programming languages required programmers to work with memory addresses and to memorize awkward codes associated with machine languages.

Newer programming languages look much more like natural language and are easier to use, partly because they allow programmers to name variables instead of using unwieldy memory addresses Initially the programs are to be written in machine language but it is in the form of 0's and First hence difficult to remember.

Second layer of assembly language which has Mnemonics in the form of English language. Language used by ALU section of the CPU.

Third layer is procedure oriented language (POP) language. In this a problem is viewed as a sequence of Instructions. All functions or tasks are combined in one procedure program.

In the fourth layer the program is divided into functions. Instructions of the program is divided into groups known as functions.

In multi function program, many important data items are placed as global so that they may be accessed by all the functions. Each function may have its own local data. In the fifth layer, modularization is used with the help of functions and in large programs it is very difficult to identify what data is used by which function. Hence data hiding concept can be provided using functions.

In object oriented, following characteristics are followed:
1. Large programs are divided into smaller programs known as functions called objects.
2. Data hiding concept is provided.

Currently, two major models or paradigms are used by programmers to develop programs and their procedures:

Procedural programming focuses on the procedures that programmers create along with modularization That is, procedural programmers focus on the actions that are carried out—for example, getting input data for an employee and writing the calculations needed to produce a paycheck from the data.

Object-oriented programming focuses on objects, or ''things,'' and describes their features (also called attributes) and behaviors.

For example, object-oriented programmers might design a payroll application by thinking about employees and paychecks, and by describing their attributes. Employees have names and Social

Security numbers, and paychecks have names and check amounts. Then the programmers would think about the behaviors of employees and paychecks, such as employees getting raises and adding dependents and paychecks being calculated and output. Object-oriented programmers would then build applications from these entities.

2.3 UNDERSTANDING THE EVOLUTION OF PROGRAMMING MODELS:

1. The oldest computer programs were written in many separate modules.
2. Procedural programmers focus on actions that are carried out by a program.
3. Object-oriented programmers focus on a program's objects and their attribute and behaviors.

Desirable Program Characteristics:

These characteristics apply to programs that are written in any programming language:-

1. Integrity:

This refers to the accuracy of the calculations. Integrity is needed to perform correct calculations if any enhancement is done otherwise there will be no use of enhancement and all enhancement will be meaningless Thus, the integrity of the calculations is an absolute necessity in any computer program.

2. Clarity:

Refers to the overall readability of the program, with specific logic. If a program should not be complicated it should be clearly written, it should be possible for another programmer to follow the program logic without much effort. It should also be possible for the original author to follow his or her own program after being away from the program for an extended period of time. One of the objectives in the design of C is the development of clear, readable and disciplined approach to programming

3. Simplicity:

The clarity readability of the program and accuracy of a program are usually enhanced by keeping things as simple as possible, uniqueness and consistency should be included with the overall program objectives. In fact, it may be desirable to sacrifice a certain amount of computational

efficiency in order to maintain a relatively simple, straightforward program structure.

4. Efficiency:

It is concerned with execution speed and efficient memory utilization. Many complex programs require a tradeoff between these characteristics. Hence experience and common sense are key factors are used to increase efficiency of the program.

5. Modularity:

Many programs can be broken down into a series of identifiable subtasks. It is good programming practice to implement each of these subtasks as a separate program module. In C programming language, such modules are written as functions. The use of a modular programming structure enhances the accuracy and clarity of a program, and it facilitates future program alterations.

6. Generality:

Program to be as general as possible, within reasonable limits. For example, we may design a program to read in the values of certain key parameters rather than placing fixed values into the program. As a rule, a considerable amount of generality can be obtained with very little additional programming effort. All programs should be written in a generalized manner.

2.3 UNIT END QUESTIONS

1. Explain Program Characteristics in Detail.

2. Draw and Explain Evolution of Programming Model.

3. Explain the process of program execution.

4. What is IDE ? Explain TurboC Functions in Detail.

*****

3

FUNDAMENTALS

Unit Structure

3.0 OBJECTIVES

Structure of a C Program:

Every C program consists of one or more modules called functions. One of the functions must be called main.

The program will always begin by executing the main function, which may access other functions. Any other function definitions must be defined separately, either ahead of or after main Each function must contain:

1. A function heading, which consists of the function name, followed by an optional list of arguments, enclosed in parentheses.

2. A list of argument declarations, if arguments are included in the heading.

3. A compound statement, which comprises the remainder of the function.

The arguments are symbols that represent information being passed between the function and other parts of the program. (Arguments are also referred to as parameters.)Each compound statement is enclosed within a pair of braces, i.e., { }. The braces may contain one or more elementary statements (called expression statements) and other compound statements. Thus compound statements may be nested, one within another. Each expression statement must end with a semicolon (;). Comments (remarks) may appear anywhere within a program, as long as they are placed within the delimiters / * and */ (e.g., /* t h i s is a comment */). Such comments are helpful in identifying the program's principal features or in explaining the underlying logic of various program features. These program components will be discussed in much greater detail later in this

book. For now, the reader should be concerned only with an overview of the basic features that characterize most C programs.

3.1 INTRODUCTION

EXAMPLE - Area of a Circle Here is an elementary C program that reads in the radius of a circle, calculates its area and then writes the calculated result.

```
/* program to calculate the area of a circle */
/* TITLE(COMMENT) */
#include <stdio.h> /          * LIBRARY FILE ACCESS */
main( ) /                     * FUNCTION HEADING */
float radius, area; /         * VARIABLE DECLARATIONS */
printf ("Radius = ? /         * OUTPUT STATEMENT (PROMPT) * /
") ;
scanf ( "%'f " , &radius) ; / * INPUT STATEMENT * /
area = 3.14159 * radius * radius; /* ASSIGNMENT STATEMENT */
printf ("Area = %f ",area) ; /  * OUTPUT STATEMENT */
```

The comments at the end of each line have been added in order to emphasize the overall program organization.

Normally a C program will not look like this. Rather, it might appear as shown below.

```
/ * program to calculate the area of a c i r c l e */
#include <stdio.h>
main( )
float radius, area;
printf ("Radius = ? " ) ;
scanf ("%f &radius) ;
area = 3.14159 * radius * radius;
pintf ("Area = %f ", area) ;
```

The following features should be pointed out in this last program.

1. The program is typed in lowercase. Either upper- or lowercase can be used, though it is customary to type ordinary instructions in lowercase. Most comments are also typed in lowercase, though comments are Sometimes typed in uppercase for emphasis, or to distinguish certain comments from the instructions.

2. The first line is a comment that identifies the purpose of the program.

3. The second line contains a reference to a special file (called stdio .h) which contains information that must be included in the program when it is compiled. The inclusion of this required information will be handled automatically by the compiler.

4. The third line is a heading for the function main. The empty parentheses following the name of the function indicate that this function does not include any arguments.

5. The remaining five lines of the program are indented and enclosed within a pair of braces. These five lines comprise the compound statement within main.

6. The first indented line is a variable declaration. It establishes the symbolic names radius and area as floating-point variables (more about this in the next chapter).

7. The remaining four indented lines are expression statements. The second indented line (p r i n t f) generates a request for information (namely, a value for the radius). This value is entered into the computer via the third indented line (scanf).

8. The fourth indented line is a particular type of expression statement called an assignment statement. This statement causes the area to be calculated from the given value of the radius. Within this statement the asterisks (*) represent multiplication signs.

9. The last indented line (p r i n t f) causes the calculated value for the area to be displayed. The numerical value will be preceded by a brief label.

10. Notice that each expression statement within the compound statement ends with a semicolon. This is required of all expression statements.

Finally, notice the liberal use of spacing and' indentation, creating whitespace within the program. The blank lines separate different parts of the program into logically identifiable components, and the indentation indicates subordinate relationships among the various instructions. These features are not grammatically essential, but their presence is strongly encouraged as a matter of good programming practice.

Execution of the program results in an interactive dialog such as that shown below. The user's response is underlined, for clarity.

Radius = 7 3

Area = 28.274309

3.2 PROGRAM STRUCTURE

Hello World Example:

A C program basically consists of the following parts:
- Preprocessor command
- Functions
- Variables

- Statements & Expressions
- Comments

Let us look at a simple code that would print the words "Hello World":

```
#include intmain()
{
/* my first program in C */
printf("Hello, World! \n");
return 0;
}
```

Let us take a look at the various parts of the above program:

1. The first line of the program #include is a preprocessor command, which tells a C compiler to include stdio.h file before going to actual compilation.

2. The next line intmain() is the main function where the program execution begins.

3. The next line /*...*/ will be ignored by the compiler and it has been put to add additional comments in the program. So such lines are called comments in the program.

4. The next line printf(...) is another function available in C which causes the message "Hello, World!" to be displayed on the screen.

5. The next line return 0; terminates the main() function and returns the value 0.

Compile and Execute C Program:

Let us see how to save the source code in a file, and how to compile and run it. Following are the simple steps:

1. Open a text editor and add the above-mentioned code.

2. Save the file as hello.c

3. Open a command prompt and go to the directory where you have saved the file.

4. Type gcchello.c and press enter to compile your code.

5. If there are no errors in your code, the command prompt will take you to the next line and would generate a.out executable file.

6. Now, type a.out to execute your program.

7. You will see the output "Hello World" printed on the screen.

$ gcchello.c

$ ./a.out

Hello, World!

Make sure the gcc compiler is in your path and that you are running it in the directory containing the source file hello.c.

3.3 THE C CHARACTER SET

C uses the uppercase letters A to Z, the lowercase letters a to z, the digits 0 to 9, and certain special characters as building blocks to form basic program elements (e.g., constants, variables, operators, expressions, etc.). The special characters are listed below.

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

Most versions of the language also allow certain other characters, such as @ and $, to be included within strings and comments.

Identifiers and Keywords:

Identifiers are names that are given to various program elements, such as variables, functions and arrays. Identifiers consist of letters and digits, in any order, except that the first character must be a letter. Both upper- and lowercase letters are permitted, though common usage favors the use of lowercase letters for most types of identifiers. Upper- and lowercase letters are not interchangeable (i.e., an uppercase letter is not equivalent to the corresponding lowercase letter.) The underscore character (-) can also be included, and is considered to be a letter. An underscore is often used in the middle of an identifier. An identifier may also begin with an underscore, though this is rarely done in practice.

EXAMPLE The following names are valid identifiers.

X Y12 sum-1 _temperature

Names area tax-r ate

TABLE

Keywords:

The following list shows the reserved words in C. These reserved words may not be used as constants or variables or any other identifier names.

Auto	char	signed	default
register	short	If	long
static	void	Float	continue
else	return	Switch	while
goto	const	Extern	unsigned
Do	sizeof	Break	typedef
int	volatile	Case	double
for	enum	Union	struct _Packed

3.4 DATA TYPES IN C LANGUAGE

Data types specify how we enter data into our programs and what type of data we enter. C language has some predefined set of data types to handle various kinds of data that we use in our program. These data types have different storage capacities.

C language supports 2 different type of data types,

Primary data types:

These are fundamental data types in C namely integer(int), floating(float), charater(char) and void.

Derived data types:

Derived data types are like arrays, functions, structures and pointers. These are discussed in detail later.

Data types in c refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

The types in C can be classified as follows –

Sr. No.	Types & Description
1	**Basic Types** They are arithmetic types and are further classified into: (a) integer types and (b) floating-point types.
2	**Enumerated types** They are again arithmetic types and they are used to define variables that can only assign certain discrete integer values throughout the program.
3	**The type void** The type specifier *void* indicates that no value is available.
4	**Derived types** They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types.

The array types and structure types are referred collectively as the aggregate types. The type of a function specifies the type of the function's return value. We will see the basic types in the following section, where as other types will be covered in the upcoming chapters.

Integer Types:

The following table provides the details of standard integer types with their storage sizes and value ranges

Type	Storage size	Value range
Char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
Int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
Short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
Long	8 bytes	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615

To get the exact size of a type or a variable on a particular platform, you can use the **sizeof** operator. The expressions *sizeof(type)* yields the storage size of the object or type in bytes. Given below is an example to get the size of various type on a machine using different constant defined in limits.h header file

```
#include  <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <float.h>
int main(intargc, char** argv) {
printf("CHAR_BIT : %d\n", CHAR_BIT);
printf("CHAR_MAX : %d\n", CHAR_MAX);
printf("CHAR_MIN : %d\n", CHAR_MIN);
printf("INT_MAX : %d\n", INT_MAX);
printf("INT_MIN : %d\n", INT_MIN);
printf("LONG_MAX : %ld\n", (long) LONG_MAX);
printf("LONG_MIN : %ld\n", (long) LONG_MIN);
printf("SCHAR_MAX : %d\n", SCHAR_MAX);
printf("SCHAR_MIN : %d\n", SCHAR_MIN);
printf("SHRT_MAX : %d\n", SHRT_MAX);
printf("SHRT_MIN : %d\n", SHRT_MIN);
printf("UCHAR_MAX : %d\n", UCHAR_MAX);
printf("UINT_MAX : %u\n", (unsigned int) UINT_MAX);
printf("ULONG_MAX : %lu\n", (unsigned long) ULONG_MAX);
printf("USHRT_MAX : %d\n", (unsigned short) USHRT_MAX);
return 0;
}
```

When you compile and execute the above program, it produces the following result on Linux −

```
CHAR_BIT       8
CHAR_MAX       127
CHAR_MIN     : -128
INT_MAX        2147483647
INT_MIN      : -2147483648
LONG_MAX     9223372036854775807
LONG_MIN     : -9223372036854775808
SCHAR_MAX    127
SCHAR_MIN    : -128
SHRT_MAX       32767
SHRT_MIN     : -32768
UCHAR_MAX    255
UINT_MAX       4294967295
ULONG_MAX    18446744073709551615
USHRT_MAX    65535
```

Floating-Point Types:
The following table provide the details of standard floating-point types with storage sizes and value ranges and their precision

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

The header file float.h defines macros that allow you to use these values and other details about the binary representation of real numbers in your programs. The following example prints the storage space taken by a float type and its range values

```
#include  <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <float.h>

int main(intargc, char** argv) {

printf("Storage size for float : %d \n", sizeof(float));
printf("FLT_MAX : %g\n", (float) FLT_MAX);
printf("FLT_MIN : %g\n", (float) FLT_MIN);
printf("-FLT_MAX : %g\n", (float) -FLT_MAX);
printf("-FLT_MIN : %g\n", (float) -FLT_MIN);
```

```
printf("DBL_MAX : %g\n", (double) DBL_MAX);
printf("DBL_MIN : %g\n", (double) DBL_MIN);
printf("-DBL_MAX : %g\n", (double) -DBL_MAX);
printf("Precision value: %d\n", FLT_DIG );

return 0;
}
```

When you compile and execute the above program, it produces the

following result on Linux −

Storage size for float : 4
FLT_MAX   : 3.40282e+38
FLT_MIN   : 1.17549e-38
-FLT_MAX   : -3.40282e+38
-FLT_MIN   : -1.17549e-38
DBL_MAX   : 1.79769e+308
DBL_MIN   : 2.22507e-308
-DBL_MAX   : -1.79769e+308
Precision value: 6

The void Type:

The void type specifies that no value is available. It is used in three kinds
of situations

Sr.No.	Types & Description
1	**Function returns as void** There are various functions in C which do not return any value or you can say they return void. A function with no return value has the return type as void. For example, **void exit (int status);**
2	**Function arguments as void** There are various functions in C which do not accept any parameter. A function with no parameter can accept a void. For example, **int rand(void);**
3	**Pointers to void** A pointer of type void * represents the address of an object, but not its type. For example, a memory allocation function **void *malloc(size_t size);** returns a pointer to void which can be casted to any data type.

```
                          Primary Data type
                                 │
        ┌────────────────────────┼────────────────────────┐
        │                        │                         │            │
   Character                 Integer                    Float         void
     ┌ char            signed │ unsigned                 ┌ float
     │                 ┌──────┴──────┐                   │
     │ Signed          │          ┌ int                  │ double
     │ char            ├ int      │                       │
     │                 │          ├ short int            └ long double
     └ Unsigned        ├ short int │
       char            └ long int └ long int
```
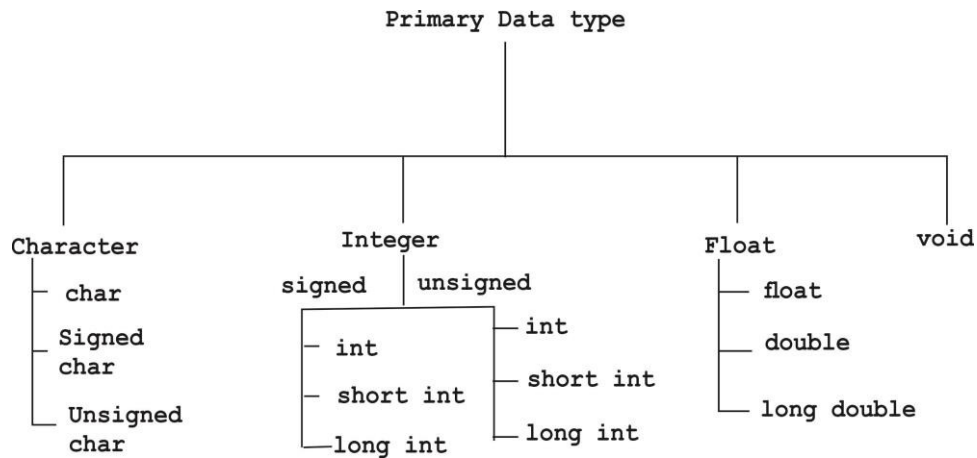
Constants:

      Constants are of fixed values that do not change during the execution of a program. There are various types of constants. The types are illustrated in the following figure.
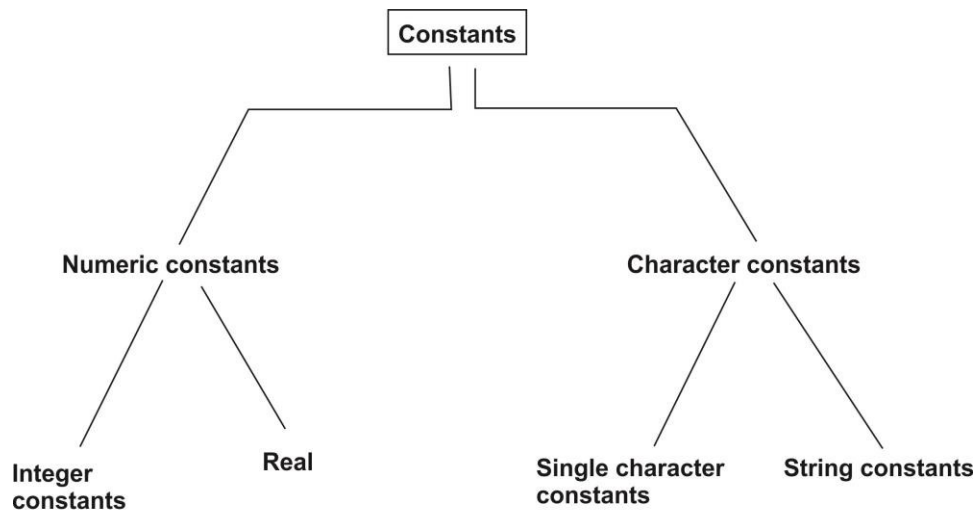
Figure : Basic types of constants

Integer constants:

      An integer constant refers to a sequence of digits. There are three types of integer constants, namely, decimal integer, octal integer and hexadecimal integer.

      Decimal integer consists of a set of digits from 0 to 9, preceded by an optional + or – sign. Examples, 123 -321 0 64932
Octal integer consists of a set of digits from 0 to 7, with a leading 0.

Examples, 037 0 0437 0551

      A sequence of digits preceded by 0x or 0X is considered as hexadecimal integer. They may also includes letters from A to F or from a to f. The letters represents the numbers from 10 to 15.

37

Examples, 0X2 0x9F
0Xbcd 0x

Real constants: Real constants are used to represent quantities that are very continuously, such as distances, temperature etc. These quantities are represented by numbers containing fractional parts.

Examples,
0.00832 -0.75 33.337
Single character constants: A single character constants contains a single character enclosed within a pair of single quote marks.

Example, _5
'X' ':'
String constants : A string constant contains a string of characters enclosed within a pair of double quote marks. Examples, ''Hello !''
''1987''

Variables:

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C is case-sensitive.

Based on the basic types explained in the previous chapter, there will be the following basic variable types:

Type	Description
Char	Typically a single octet (one byte). This is an integer type.
int	The most natural size of integer for the machine
float	A single-precision floating point value.
double	A double-precision floating point value
void	Represents the absence of type. C

Variable Definition in C:
A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows:

typevariable_list;

Here, type must be a valid C data type including char, w_char, int, float, double, bool, or any user-defined object; and variable_list may consist of one or more identifier names separated by commas. Some valid declarations are shown here: Variable Definition in C :A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows:

inti, j, k;

char c, ch;

float f salary;

double d;

The line int i, j, k; declares and defines the variables i, j and k; which instruct the compiler to create variables named i, j, and k of type int.

Variables can be initialized (assigned an initial value) in their declaration. The initialize consists of an equal sign followed by a constant expression as follows:

typevariable_name = value;

Some examples are:

Extern int d = 3, f = 5; // declaration of d and f.

int d = 3, f = 5; // definition and initializing d and f.

byte z = 22; // definition and initializes z.

char x = 'x'; // the variable x has the value 'x'.

C –Arrays:

Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.
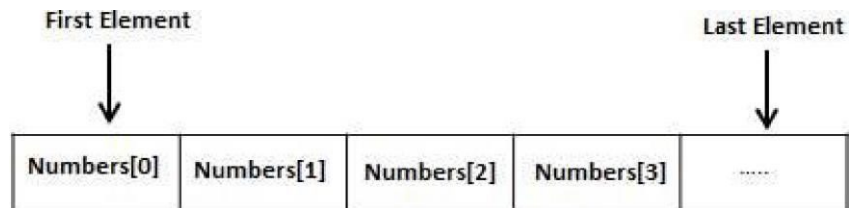
Declaring Arrays :

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows −

typearrayName [arraySize];

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement −

double balance[10];
Here *balance* is a variable array which is sufficient to hold up to 10 double numbers.

Initializing Arrays:

You can initialize an array in C either one by one or using a single statement as follows −

double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array −

40

balance[4] = 50.0;

The above statement assigns the 5th element in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the array we discussed above –

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

Accessing Array Elements:

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example −

double salary = balance[9];

The above statement will take the 10th element from the array and assign the value to salary variable. The following example Shows how to use all the three above mentioned concepts viz. declaration, assignment, and accessing arrays –

```
#include <stdio.h>

int main () {

int n[ 10 ]; /* n is an array of 10 integers */
inti,j;

/* initialize elements of array n to 0 */
for ( i = 0; i< 10; i++ ) {
n[ i ] = i + 100; /* set element at location i to i + 100 */
}

/* output each array element's value */
for (j = 0; j < 10; j++ ) {
printf("Element[%d] = %d\n", j, n[j] );
}

return 0;
}
```

When the above code is compiled and executed, it produces the following result −

Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109

Arrays in Detail:

Arrays are important to C and should need a lot more attention. The following important concepts related to array should be clear to a C programmer

Sr.No.	Concept & Description
1	**Multi-dimensional arrays** C supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
2	**Passing arrays to functions** You can pass to the function a pointer to an array by specifying the array's name without an index.
4	**Pointer to an array** You can generate a pointer to the first element of an array by simply specifying the array name, without any index.

3.5 C EXPRESSIONS

An expression is a formula in which operands are linked to each other by the use of operators to compute a value. An operand can be a function reference, a variable, an array element or a constant.

Let's see an example:
1. a-b;

In the above expression, minus character (-) is an operator, and a, and b are the two operands.

There are four types of expressions exist in C:

Arithmetic expressions
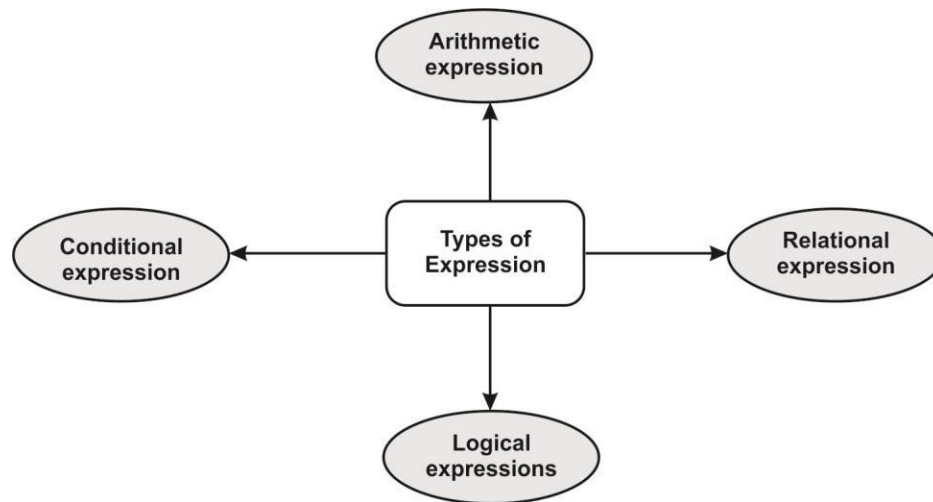- Relational expressions
- Logical expressions

- Conditional expressions

Each type of expression takes certain types of operands and uses a specific set of operators. Evaluation of a particular expression produces a specific value.

For example:

1. x = 9/2 + a-b;

The entire above line is a statement, not an expression. The portion after the equal is an expression.

Arithmetic Expressions:

An arithmetic expression is an expression that consists of operands and arithmetic operators. An arithmetic expression computes a value of type int, float or double.

When an expression contains only integral operands, then it is known as pure integer expression when it contains only real operands, it is known as pure real expression, and when it contains both integral and real operands, it is known as mixed mode expression.

Evaluation of Arithmetic Expressions:

The expressions are evaluated by performing one operation at a time. The precedence and associativity of operators decide the order of the evaluation of individual operations.

When individual operations are performed, the following cases can be happened:

- When both the operands are of type integer, then arithmetic will be performed, and the result of the operation would be an integer value. For example, 3/2 will yield 1 not 1.5 as the fractional part is ignored.

- When both the operands are of type float, then arithmetic will be performed, and the result of the operation would be a real value. For example, 2.0/2.0 will yield 1.0, not 1.

- If one operand is of type integer and another operand is of type real, then the mixed arithmetic will be performed. In this case, the first operand is converted into a real operand, and then arithmetic is performed to produce the real value. For example, 6/2.0 will yield 3.0 as the first value of 6 is converted into 6.0 and then arithmetic is performed to produce 3.0.

Let's understand through an example.

6*2/ (2+1 * 2/3 + 6) + 8 * (8/4)

Evaluation of expression	Description of each operation
6*2/(2+1 * 2/3 +6) +8 * (8/4)	An expression is given
6*2/(2+2/3 + 6) + 8 * (8/4)	2 is multiplied by 1, giving value 2.
6*2/(2+0+6) + 8 * (8/4)	2 is divided by 3, giving value 0.
6*2/ 8+ 8 * (8/4)	2 is added to 6, giving value 8.
6*2/8 + 8 * 2	8 is divided by 4, giving value 2.
12/8 +8 * 2	6 is multiplied by 2, giving value 12.
1 + 8 * 2	12 is divided by 8, giving value 1.
1 + 16	8 is multiplied by 2, giving value 16.
17	1 is added to 16, giving value 17.

Relational Expressions:

- A relational expression is an expression used to compare two operands.

- It is a condition which is used to decide whether the action should be taken or not.

- In relational expressions, a numeric value cannot be compared with the string value.

- The result of the relational expression can be either zero or non-zero value. Here, the zero value is equivalent to a false and non-zero value is equivalent to true.

Relational Expression	Description
x%2 = = 0	This condition is used to check whether the x is an even number or not.
	The relational expression results in value 1 if x is an even number otherwise results in value 0.
a!=b	It is used to check whether a is not equal to b.

	This relational expression results in 1 if a is not equal to b otherwise 0.
a+b = = x+y	It is used to check whether the expression "a+b" is equal to the expression "x+y".
a>=9	It is used to check whether the value of a is greater than or equal to 9.
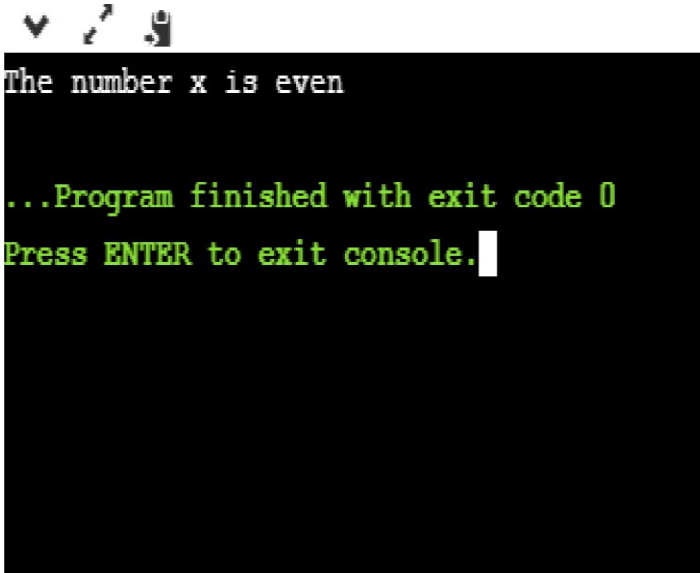
Let's see a simple example:
1. #include <stdio.h>
2. **int** main()
3. {
4.
5. **int** x=4;
6. **if**(x%2==0)
7. {
8. printf("The number x is even");
9. }
10. **else**
11. printf("The number x is not even");
12. **return** 0;
13. }

Output :


```
The number x is even


...Program finished with exit code 0
Press ENTER to exit console.
```

Logical Expressions:
- A logical expression is an expression that computes either a zero or non-zero value.
- It is a complex test condition to take a decision.

Let's see some example of the logical expressions.

Logical Expressions	Description
(x > 4) && (x < 6)	It is a test condition to check
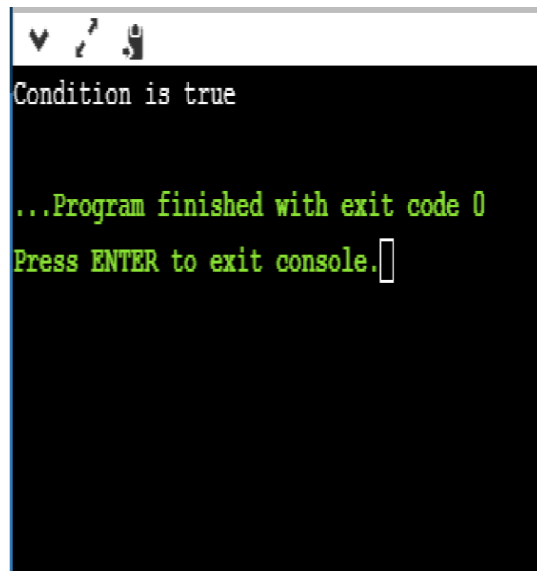
	whether the x is greater than 4 and x is less than 6. The result of the condition is true only when both the conditions are true.
x > 10 || y <11	It is a test condition used to check whether x is greater than 10 or y is less than 11. The result of the test condition is true if either of the conditions holds true value.
! (x > 10) && (y = = 2)	It is a test condition used to check whether x is not greater than 10 and y is equal to 2. The result of the condition is true if both the conditions are true.

Let's see a simple program of "&&" operator.
1.   #include <stdio.h>
2.   **int** main()
3.   {
4.   **int** x = 4;
5.   **int** y = 10;
6.   **if** ((x <10) && (y>5))
7.   {
8.   printf("Condition is true");
9.   }
10.   **else**
11.   printf("Condition is false");
12.   **return** 0;
13.   }

Output

Let's see a simple example of "||" operator

46

```
1. #include <stdio.h>
2. int main()
3. {
4. int x = 4;
5. int y = 9;
6. if ( (x <6) || (y>10))
7. {
8. printf("Condition is true");
9. }
10. else
11. printf("Condition is false");
12. return 0;
13. }
```
Output

```
Condition is true

...Program finished with exit code 0
Press ENTER to exit console.
```

Conditional Expressions:

- A conditional expression is an expression that returns 1 if the condition is true otherwise 0.

- A conditional operator is also known as a ternary operator.

The Syntax of Conditional operator:

Suppose exp1, exp2 and exp3 are three expressions.

exp1 ?exp2 : exp3

The above expression is a conditional expression which is evaluated on the basis of the value of the exp1 expression. If the condition of the expression exp1 holds true, then the final conditional expression is represented by exp2 otherwise represented by exp3.

Let's understand through a simple example.

```
1. #include<stdio.h>
2. #include<string.h>
3. int main()
4. {
5. int age = 25;
```

47
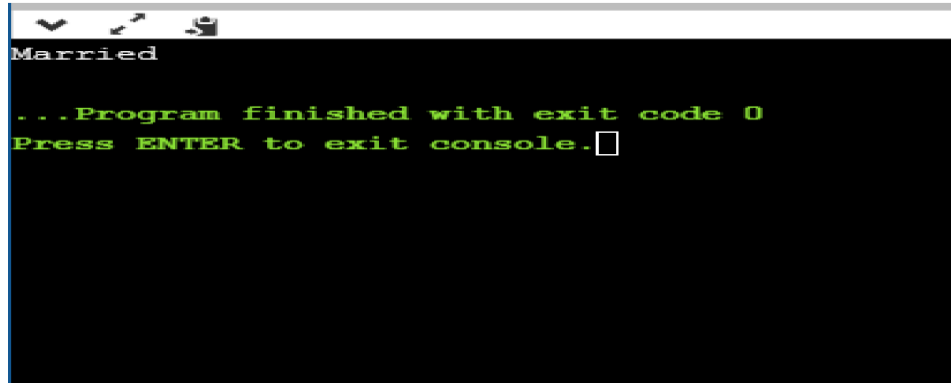
6. **char** status;
7. status = (age>22) ? 'M': 'U';
8. **if**(status == 'M')
9. printf("Married");
10. **else**
11. printf("Unmarried");
12. **return** 0;
13. }

Output:


```
Married

...Program finished with exit code 0
Press ENTER to exit console.
```

Statements:

A statement causes the computer to carry out some action. There are three different classes of statements in C.

They are expression statements, compound statements and control statements.

An expression statement consists of an expression followed by a semicolon. The execution of an expression statement causes the expression to be evaluated.

EXAMPLE 2.28 Several expression statements are shown below.

a = 3;
c=a+b;
++i;
printf ("Area = %f 'I, area) ;
9

The first two expression statements are assignment-type statements. Each causes the value of the expression on the right of the equal sign to be assigned to the variable on the left. The third expression statement is an incrementing-type statement, which causes the value of ito increase by 1.

The fourth expression statement causes the printf function to be evaluated. This is a standard C library function that writes information out of the computer (more about this in Sec. 3.6). In this case, the message Area = will be displayed, followed by the current value of the variable area. Thus, if area represents the value loo., the statement will generate the message

Area = 100.

The last expression statement does nothing, since it consists of only a semicolon. It is simply a mechanism for providing an empty expression statement in places where this type of statement is required. Consequently, it is called a null statement.

A compound statement consists of several individual statements enclosed within a pair of braces { }.

The individual statements may themselves be expression statements, compound statements or control statements. Thus, the compound statement provides a capability for embedding statements within other statements. Unlike an expression statement, a compound statement does not end with a semicolon.

EXAMPLE - A typical compound statement is shown below.
pi = 3.141593;
circumference = 2. * pi * radius;
area = pi * radius * radius;

This particular compound statement consists of three assignment-type expression statements, though it is considered a single entity within the program in which it appears. Note that the compound statement does not end with a semicolon &er the brace.

3.6 SYMBOLIC CONSTANTS IN C LANGUAGE

● When a constant is used at many places in a program ,due to some reason if the value of that constant needs to be changed,then change at every statement where that constant occurs in the program –so modification of program becomes difficult.

● Symbolic constant is defined as below:
#define symbolic_name value
Example :
#define FLAG 1
#define PI 3.1415

Here, FLAG and PI are symbolic constants. For better readability, it is advisable to use uppercase character in the naming of symbolic constants. see there is no semicolon ';' at the end.

Program:

```
/* program illustrating use of declaration,assignment of value to
variables.also explains how to use symbolic constants.

program to calculate area and circumference of a circle */

#include<stdio.h>

#include<conio.h>

#define PI 3.1415 /* NO SEMICOLON HERE */

void main ()

{

float rad = 5; /*DECLARATION AND ASSIGNMENT*/
float area,circum; /* DECLARATION OF VARIABLE*/
area=PI*rad*rad;
circum=2*PI*rad;
printf(―AREA OF CIRCLE = %f\n",area);
printf(―CIRCUMFERENCE OF CIRCLE =%f\n",circum);
getch();
clrscr();

}
OUTPUT :
```
```
AREA OF CIRCLE =78.537498
CIRCUMFERENCE OF CIRCLE =31.415001
```

3.7 UNIT END QUESTIONS

1. What is Charset? Explain with Example.

2. What is Keyword? Explain with Example.

3. What is the Use of sizeof() method ? Explain with Example.

4. Draw and Explain Hierarchy of DataTypes.

5. What is Variable? Explain with Example.

6. What is Expression? Explain Arithmetic Expression in Detail.

*****

OPERATORS AND EXPRESSIONS - I

Unit Structure

4I.0 Objectives

4I.1 Introduction

4I.2 Unary Operators

4I.3 Unit End Questions

4I.0 OBJECTIVES

We have already seen that individual constants, variables, array elements and function references can be joined together by various operators to form expressions. We have also mentioned that C includes a large number of operators which fall into several different categories. In this chapter we examine certain of these categories in detail. Specifically, we will see how arithmetic operators, unary operators, relational and logical operators, assignment operators and the conditional operator are used to form expressions. The data items that operators act upon are called operands. Some operators require two operands, while others act upon only one operand. Most operators allow the individual operands to be expressions. A few operators permit only single variables as operands (more about this later).

4I.1 INTRODUCTION

ARITHMETIC OPERATORS:
There are five arithmetic operators in C. They are

Operator	Purpose
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder after integer division

The % operator is sometimes referred to as the modulus operator.

There is no exponentiation operator in C. However, there is a library function (POW) to carry out exponentiation. The operands acted upon by arithmetic operators must represent numeric values. Thus, the operands can be integer quantities, floating-point quantities or characters (remember that character constants represent integer values, as determined

by the computer's character set). The remainder operator (%) requires that both operands be integers and the second operand be nonzero. Similarly, the division operator (/) requires that the second operand be nonzero.

Division of one integer quantity by another is referred to as integer division. This operation always results in a truncated quotient (i.e., the decimal portion of the quotient will be dropped). On the other hand, if a division operation is carried out with two floating-point numbers, or with one floating-point number and one integer, the result will be a floating-point quotient.

EXAMPLE - Suppose that A and B are integer variables whose values are 10 and 3, respectively. Several arithmetic expressions involving these variables - are shown below, together with their resulting values.

Expression	Value
A + B	13
A − B	7
A * B	30
A / B	3
A % B	1

Notice the truncated quotient resulting from the division operation, since both operands represent integer quantities.

Also, notice the integer remainder resulting from the use of the modulus operator in the last expression.

Now suppose that A and B are floating-point variables whose values are 12.5 and 2.0, respectively. Several - arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	Value
A + B	14.5
A − B	10.5
A * B	25.0
A / B	6.25

4I.2 UNARY OPERATORS

C includes a class of operators that act upon a single operand to produce a new value. Such operators are known as unary operators. Unary operators usually precede their single operands, though some unary operators are written after their operands.

Perhaps the most common unary operation is unary minus, where a numerical constant, variable or expression is preceded by a minus sign. (Some programming languages allow a minus sign to be included as a part

of a numeric constant. In C, however, all numeric constants are positive. Thus, a negative number is actually an expression, consisting of the unary minus operator, followed by a positive numeric constant.)

Note that the unary minus operation is distinctly different from the arithmetic operator which denotes subtraction (-). The subtraction operator requires two separate operands.

EXAMPLE 3.10 Here are several examples which illustrate the use of the unary minus operation

-743	-0X7FFF	-0.2	-5E-8
-root1	-(x + Y)	-3 *(x + y)	

In each case the minus sign is followed by a numerical operand which may be an integer constant, a floating-point constant, a numeric variable or an arithmetic expression.

There are two other commonly used unary operators: The increment operator, ++, and the decrement operator, —. The increment operator causes its operand to be increased by 1, whereas the decrement operator causes its operand to be decreased by 1. The operand used with each of these operators must be a single variable.

Relational And Logical Operators:
There are four relational operators in C. They are

Operator	Meaning
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

These operators all fall within the same precedence group, which is lower than the arithmetic and unary operators. The associativity of these operators is left to right.

Closely associated with the relational operators are the following two equality operators,

Operator	Meaning
= =	equal to
!=	not equal to

The equality operators fall into a separate precedence group, beneath the relational operators. These operators also have a left-to-right associativity. These six operators are used to form logical expressions, which represent conditions that are either true or false. The resulting

53

expressions will be of type integer, since true is represented by the integer value 1 and false is represented by the value 0.

EXAMPLE - Suppose that i, j and k are integer variables whose values are 1, 2 and 3, respectively. Several logical expressions involving these variables are shown below.

Expression	Intecmetation	Value
I < j	True	1
(i+j) >=k	True	1
(j + k) > (i + 5)	False	0
k I= 3	False	0
j == 2	True	1

In addition to the relational and equality operators, C contains two logical operators (also called logical connectives). They are

Operator	Meaning
&&	And
II	Or

These operators are referred to as logical and and logical or, respectively.

The logical operators act upon operands that are themselves logical expressions. The net effect is to combine the individual logical expressions into more complex conditions that are either true or false. The result of a logical and operation will be true only if both operands are true, whereas the result of a logical or operation will be true if either operand is true or if both operands are true. In other words, the result of a logical or operation will be false only if both operands are false.

In this context it should be pointed out that any nonzero value, not just 1, is interpreted as true.

EXAMPLE - Suppose that i is an integer variable whose value is 7, f is a floating-point variable whose value is 5.5, and c is a character variable that represents the character ' w ' . Several complex logical expressions that make use of these variables are shown below.

Expression	Interpretation	Value
(i >= 6) && (c == 'w')	True	1
(i >= 6) 11 (c == 119	True	1
(f < 11) && (i > 100)	False	0
(c != 'p') II ((i + f) <= 10)	True	1

The first expression is true because both operands are true. In the second expression, both operands are again true; hence the overall expression is true. The third expression is false because the second

operand is false. And finally, the fourth expression is true because the first operand is true.

Assignment Operators :

There are several different assignment operators in C. All of them are used to form assignment expressions, which assign the value of an expression to an identifier. \

The most commonly used assignment operator is =.
Assignment expressions that make use of this operator are written in the form
identifier = expression

where identifier generally represents a variable, and expression represents a constant, a variable or a more complex expression.

EXAMPLE - Here are some typical assignment expressions that make use of the = operator.
a=3

x=y

delta = 0.001

sum = a + b

area = length * width

The first assignment expression causes the integer value 3 to be assigned to the variable a, and the second assignment causes the value of y to be assigned to x. In the third assignment, the floating-point value 0.001 is assigned to delta.

The last two assignments each result in the value of an arithmetic expression being assigned to a variable (i.e., the value of a + b is assigned to sum, and the value of length * width is assigned to area).

The Conditional Operator:

Simple conditional operations can be carried out with the conditional operator (? :). An expression that makes use of the conditional operator is called a conditional expression. Such an expression can be written in place of the more traditional if -else statement.

A conditional expression is written in the form

expression 1 ? expression 2 : expression 3

When evaluating a conditional expression, expression 1 is evaluated first. If expression 1 is true (i.e., if its value is nonzero), then expression 2 is evaluated and this becomes the value of the conditional expression. However, if expression 1 is false (i.e., if its value is zero), then expression 3is evaluated and this becomes the value of the conditional expression. Note that only one of the embedded expressions (either expression 2 or expression 3) is evaluated when determining the value of a conditional expression.

EXAMPLE: In the conditional expression shown below, assume that i is an integer variable.
(i < 0) ? 0 : 100

The expression (i < 0) is evaluated first. If it is true (i.e., if the value of i is less than 0), the entire conditional expression takes on the value 0. Otherwise (if the value of i is not less than 0),the entire conditional expression takes on the value 100.

In the following conditional expression, assume that f and g are floating-point variables.

(f<g)?f :g

This conditional expression takes on the value off if f is less than g; otherwise, the conditional expression takes on the value of g. In other words, the conditional expression returns the value of the smaller of the two variables.

4I.3 UNIT END QUESTIONS

1. What is Operator ? Explain Arithmetic Operators in Detail.
2. What is Operator ? Explain Assignment Operators in Detail.
3. Write program to Create Calculator (+ - * /) in C
4. Explain conditional operator (? :) with Example.

4II

OPERATORS AND EXPRESSIONS – II

Unit Structure

4II.0 Objectives

4II.1 Introduction

4II.2 Operators Precedence In C

4II.2 Unit End Questions

4II.0 OBJECTIVES

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators and provides the following types of operators

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

We will, in this chapter, look into the way each operator works.

4II.1 INTRODUCTION

Arithmetic Operators:

The following table shows all the arithmetic operators supported by the C language. Assume variable A holds 20 and variable B holds 30 then

Operator	Description	Example
+	Adds two operands.	A + B = 50
-	Subtracts second operand from the first.	A - B = -10
*	Multiplies both operands.	A * B = 600
/	Divides numerator by de-numerator.	B / A = 1.5
%	Modulus Operator and remainder of after an integer division.	B % A = 0

++	Increment operator increases the integer value by one.	A++ = 21
—	Decrement operator decreases the integer value by one.	14 A-- = 9

Relational Operators:

The following table shows all the relational operators supported by C. Assume variable A holds 20 and variable B holds 30 then

Operator	Description	Example
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

Logical Operators:

Following table shows all the logical operators supported by C language. Assume variable A holds 1 and variable B holds 0, then

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

Bitwise Operators:

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &|, and ^ is as follows

p	Q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume A = 60 and B = 13 in binary format, they will be as follows "
A = 0011 1100
B = 0000 1101

_____

A&B = 0000 1100
A|B = 0011 1101
A^B = 0011 0001
~A = 1100 0011

The following table lists the bitwise operators supported by C. Assume variable 'A' holds 60 and variable 'B' holds 13, then

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand	(A | B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = ~(60), i.e,. -0111101
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

Assignment Operators:

The following table lists the assignment operators supported by the C language.

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A

%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Misc Operators ↦ sizeof & ternary:

Besides the operators discussed above, there are a few other important operators including sizeof and ? : supported by the C Language.

Operator	Description	Example
sizeof()	Returns the size of a variable.	sizeof(a), where a is integer, will return 4.
&	Returns the address of a variable.	&a; returns the actual address of the variable.
*	Pointer to a variable.	*a;
? :	Conditional Expression.	If Condition is true ? then value X : otherwise value Y

4II.2 OPERATORS PRECEDENCE IN C

Operator precedence determines the grouping of terms in an expression and decides how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has a higher precedence than the addition operator.

For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has a higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left

Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
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	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left

Arithmetic Operator:
Example:

```
#include <stdio.h>

main() {

int a = 21;
int b = 10;
int c ;

c = a + b;
printf("Value of c is %d\n", c );
c = a - b;
printf("Value of c is %d\n", c );

c = a * b;
printf("Value of c is %d\n", c );

c = a / b;
printf("Value of c is %d\n", c );

c = a % b;
printf("Value of c is %d\n", c );

c = a++;
printf("Value of c is %d\n", c );

c = a—;
printf("Value of c is %d\n", c );

}
```

Result :

Value of c is 31

Value of c is 11
Value of c is 210
Value of c is 2
Value of c is 1
Value of c is 21
Value of c is 22

Relational Operator

Example:

```
#include <stdio.h>

main() {

int a = 21;
int b = 10;
int c ;

if( a == b ) {
printf("a is equal to b\n" );
} else {
printf("a is not equal to b\n" );
}

if ( a < b ) {
printf("a is less than b\n" );
} else {
printf("a is not less than b\n" );
}

if ( a > b ) {
printf("a is greater than b\n" );
} else {
printf("a is not greater than b\n" );
}

/* Lets change value of a and b */
a = 5;
b = 20;

if ( a <= b ) {
printf("a is either less than or equal to b\n" );
}

if ( b >= a ) {
printf("b is either greater than or equal to b\n" );
}
}
```

Result:

a is not equal to b

a is not less than b

a is greater than b

a is either less than or equal to b

b is either greater than or equal to b

Logical Operator

Example :

```
#include <stdio.h>

main() {

int a = 5;
int b = 20;
int c ;

if ( a && b ) {
printf("Condition is true\n" );
}

if ( a || b ) {
printf("Condition is true\n" );
}
/* lets change the value of a and b */
a = 0;
b = 10;

if ( a && b ) {
printf("Condition is true\n" );
} else {
printf("Condition is not true\n" );
}

if ( !(a && b) ) {
printf("Condition is true\n" );
}


}
```

Result :

Condition is true

Condition is true

Bitwise Operator

Example:

```
#include <stdio.h>

main() {

unsigned int a = 60;            /* 60 = 0011 1100 */
unsigned int b = 13;            /* 13 = 0000 1101 */
int c = 0;

c = a & b;         /* 12 = 0000 1100 */
printf("Value of c is %d\n", c );

c = a | b;         /* 61 = 0011 1101 */
printf("Value of c is %d\n", c );

c = a ^ b;         /* 49 = 0011 0001 */
printf("Value of c is %d\n", c );

c = ~a;            /*-61 = 1100 0011 */
printf("Value of c is %d\n", c );

c = a << 2;        /* 240 = 1111 0000 */
printf("Value of c is %d\n", c );

c = a >> 2;        /* 15 = 0000 1111 */
printf("Value of c is %d\n", c );
}
```

Result:

Value of c is 12

Value of c is 61

Value of c is 49

Value of c is -61

Value of c is 240

Value of c is 15

Assignment Operator

Example:

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

int a = 21;
int c ;

c = a;
```

```c
    printf("= Operator Example, Value of c = %d\n", c );

    c += a;
    printf("+= Operator Example, Value of c = %d\n", c );

    c -= a;
    printf("-= Operator Example, Value of c = %d\n", c );

    c *= a;
    printf("*= Operator Example, Value of c = %d\n", c );

    c /= a;
    printf("/= Operator Example, Value of c = %d\n", c );

    c = 200;
    c %= a;
    printf("%= Operator Example, Value of c = %d\n", c );

    c <<= 2;
    printf("<<= Operator Example, Value of c = %d\n", c );

    c >>= 2;
    printf(">>= Operator Example, Value of c = %d\n", c );

    c &= 2;
    printf("&= Operator Example, Value of c = %d\n", c );

    c ^= 2;
    printf("^= Operator Example, Value of c = %d\n", c );

    c |= 2;
    printf("|= Operator Example, Value of c = %d\n", c );

}
```

Result:

= Operator Example,	Value of	c	=	21
+=	Operator Example,	Value of	c	=	42
-=	Operator Example,	Value of	c	=	21
*=	Operator Example,	Value of	c	=	441
/= Operator Example,	Value of	c	= 21
%=	Operator Example,	Value of	c	=	11
<<=	Operator Example,	Value of	c	=	44
>>=	Operator Example,	Value of	c	=	11
&=	Operator Example,	Value of	c	=	2
^=	Operator Example,	Value of	c	=	0
|= Operator Example,		Value of	c	=	2

Size of and ternary operator

Example:

```c
#include <stdio.h>

main() {

int a = 4;
short b;
double c;
int* ptr;

/* example of sizeof operator */
printf("Size of variable a = %d\n", sizeof(a) );
printf("Size of variable b = %d\n", sizeof(b) );
printf("Size of variable c= %d\n", sizeof(c) );

/* example of & and * operators */
ptr = &a; /* 'ptr' now contains the address of 'a'*/
printf("value of a is %d\n", a);
printf("*ptr is %d.\n", *ptr);

/* example of ternary operator */
a = 10;
b = (a == 1) ? 20: 30;
printf( "Value of b is %d\n", b );

b = (a == 10) ? 20: 30;
printf( "Value of b is %d\n", b );

}
```

Result:

Size of variable	a = 4
Size of variable	b = 2
Size of variable	c= 8
value of	a is 4
*ptr is 4.	
Value of	b is 30
Value of	b is 20

Operators Precedence

Example:

```c
#include <stdio.h>
main() { int a = 20;
int b = 10;
int c = 15;
int d = 5;
int e;
```

```
e = (a + b) * c / d; // ( 30 * 15 ) / 5
printf("Value of (a + b) * c / d is : %d\n", e );
e = ((a + b) * c) / d; // (30 * 15 ) / 5
printf("Value of ((a + b) * c) / d is : %d\n" , e );
e = (a + b) * (c / d); // (30) * (15/5)
printf("Value of (a + b) * (c / d) is : %d\n", e );
e = a + (b * c) / d; // 20 + (150/5)
printf("Value of a + (b * c) / d is : %d\n" , e );
return 0;
}
```

Result:

Value of (a + b)   * c / d is : 90

Value of ((a + b)   * c) / d is : 90

Value of (a + b)   * (c / d) is : 90

Value of a + (b   * c) / d is : 50

C – Library functions:

- Library functions in C language are inbuilt functions which are grouped together and placed in a common place called library.

- Each library function in C performs specific operation.

- We can make use of these library functions to get the pre-defined output instead of writing our own code to get those outputs.

- These library functions are created by the persons who designed and created C compilers.

- All C standard library functions are declared in many header files which are saved as file_name.h.

- Actually, function declaration, definition for macros are given in all header files.

- We are including these header files in our C program using "#include<file_name.h>" command to make use of the functions those are declared in the header files.

- When we include header files in our C program using "#include<filename.h>" command, all C code of the header files are included in C program. Then, this C program is compiled by compiler and executed.

List Of Most Used Header Files In C Programming Language:

- Check the below table to know all the C library functions and header files in which they are declared.

- Click on the each header file name below to know the list of inbuilt functions declared inside them.

Header file	Description
stdio.h	This is standard input/output header file in which Input/Output functions are declared
conio.h	This is console input/output header file

string.h	All string related functions are defined in this header file
stdlib.h	This header file contains general functions used in C programs
math.h	All maths related functions are defined in this header file
time.h	This header file contains time and clock related functions
ctype.h	All character handling functions are defined in this header file
stdarg.h	Variable argument functions are declared in this header file
signal.h	Signal handling functions are declared in this file
setjmp.h	This file contains all jump functions
locale.h	This file contains locale functions
errno.h	Error handling functions are given in this file
assert.h	This contains diagnostics functions

LIBRARY FUNCTIONS

The C language is accompanied by a number of library functions that carry out various commonly used operations or calculations. These library functions are not a part of the language per se, though all implementations of the language include them. Some functions return a data item to their access point; others indicate whether a condition is true or false by returning a 1 or a 0, respectively; still others carry out specific operations on data items but do not return anything. Features which tend to be computer-dependent are generally written as library functions.

For example, there are library functions that carry out standard input/output operations (e.g., read and write characters, read and write numbers, open and close files, test for end of file, etc.), functions that perform operations on characters (e.g., convert from lower- to uppercase, test to see if a character is uppercase, etc.), functions that perform operations on strings (e.g., copy a string, compare strings, concatenate strings, etc.), and functions that carry out various mathematical calculations (e.g., evaluate trigonometric, logarithmic and exponential functions, compute absolute values, square roots, etc.). Other kinds of library functions are also available.

Library functions that are functionally similar are usually grouped together as (compiled) object programs in separate library files. These library files are supplied as a part of each C compiler. All C compilers contain similar groups of library functions, though they lack precise standardization. Thus there may be some variation in the library functions that are available in different versions of the language.

A typical set of library functions will include a fairly large number of functions that are common to most C compilers, such as those shown in Table 3-2 below. Within this table, the column labeled "type" refers to the

data type of the quantity that is returned by the function. The void entry shown for function and indicates that nothing is returned by this function.

A library function is accessed simply by writing the function name, followed by a list of arguments that represent information being passed to the function. The arguments must be enclosed in parentheses and separated by commas. The arguments can be constants, variable names, or more complex expressions. The parentheses must be present, even if there are no arguments.

A function that returns a data item can appear anywhere within an expression, in place of a constant or an identifier (Le., in place of a variable or an array element). A function that carries out operations on data items but does not return anything can be accessed simply by writing the function name, since this type of function reference constitutes an expression statement.

Function	Type	Purpose
abs (i)	Int	Return the absolute value of i.
ceil (d)	double	Round up to the next integer value (the smallest integer that is greater than or equal to d).
cos (d)	double	Return the cosine of d.
cosh (d)	double	Return the hyperbolic cosine of d.
exp(d)	double	Raise e to the power d (e = 2.7182818 * is the base of the natural (Naperian) system of logarithms).
fabs (d)	double	Return the absolute value of d.
floor (d)	double	Round down to the next integer value (the largest integer that does not exceed d).
fmod (d1,d2)	double	Return the remainder (i.e., the noninteger part of the quotient) of d1 /d2, with same sign as d1 .
Getchar()	Int	Enter a character from the standard input device.
log (d)	double	Return the natural logarithm of d.
pow(d1,d2)	double	Return dl raised to the d2 power.
printf(...)	Int	Send data items to the standard output device
putchar(c)	Int	Send a character to the standard output device.
rand( )	Int	Return a random positive integer.
sin (d)	double	Return the sine of d.
sqrt(d)	double	Return the square root of d
srand (u)	Void	Initialize the random number generator
scanf(...)	Int	Enter data items from the standard input device

tan (d)	double	Return the tangent of d.
toascii(c)	Int	Convert value of argument to ASCII.
tolower (c)	Int	Convert letter to lowercase
toupper(c) .	Int	Convert letter to uppercase

Note: Type refers to the data type of the quantity that is returned by the function

c - denotes a character-type argument

i - denotes an integer argument

d - denotes a double-precision argument

u - denotes an unsigned integer argument

EXAMPLE - Lowercase to Uppercase Character Conversion Here is a complete C program that reads in a lowercase character, converts it to uppercase and then displays the uppercase equivalent.

```
/* read a lowercase character and display its uppercase equivalent */
#include <stdio.h>
#include <ctype.h>
main ( )
{
int lower, upper;
lower = getchar();
upper = toupper(lower);
putchar(upper);
}
```

This program contains three library functions: getchar, toupper and putchar. The first two functions each return a single character (getchar returns a character that is entered from the keyboard, and to upper returns the uppercase equivalent of its argument). The last function (putchar) causes the value of the argument to be displayed.

Notice that the last two functions each have one argument but the first function does not have any arguments, as indicated by the empty parentheses.

Also, notice the preprocessor statements #include <stdio. h> and #include <ctype. h>, which appear at the start of the program. These statements cause the contents of the files stdio. h and ctype .h to be inserted into the program the compilation process begins. The information contained in these files is essential for the proper functioning of the library functions getchar, putchar and toupper.

4.II UNIT END QUESTIONS

1. What is the use of sizeof & ternary operator ?
2. Explain Operators Precedence in C .
3. Write a Program to Demonstrate Bitwise Operator
4. Write a Program to Demonstrate Logical Operator
5. What are the Library functions ? Explain with example.

*****

5

DATA INPUT AND OUTPUT

Unit Structure

5.0 Objectives

5.1 Introduction

5.2 Unit End Questions

5.0 OBJECTIVES

We have already seen that the C language is accompanied by a collection of library functions, which includes a number of input output functions. In this chapter we will make use of six of these functions: getchar, putchar, scanf, printf, gets and puts. These six fuctions permit the transfer of information between the computer and the standard input output devices (e.g., a keyboard and a TV monitor). The first two functions, getchar and putchar, allow single characters to be transferred into and out of the computer; scanf and printf are the most complicated, but they permit the transfer of single characters, numerical values and strings; gets and puts facilitate the input and output of strings. Once we have learned how to use these functions, we will be able to write a number of complete, though simple, C programs.

An input output function can be accessed from anywhere within a program simply by writing the function name, followed by a list of arguments enclosed in parentheses. The arguments represent data items that are sent to the function. Some input output functions do not require arguments, though the empty parentheses must still appear.

The names of those functions that return data items may appear within expressions, as though each function reference were an ordinary variable (e.g., c = getchar ( ) ;), or they may be referenced as separate statements (e.g., scanf (. . .) ;). Some functions do not return any data items. Such functions are referenced as though they were separate statements (e.g., putchar (. . .) ;).

5.1 INTRODUCTION

EXAMPLE - Here is an outline of a typical C program that makes use of several input output routines from the standard C library.

/* sample setup illustrating the use of input/output library functions */

```
#include <stdio.h>

main ( )

{
char c,d;                       /* declarations *I

float x,y;

int i , j ,0 k ;

c = getchar();                  /* character input */

scanf ("%f", &x) ;                  /* floating-point input */

scanf ("%d %d" , &i,&j) ;           /* integer input */

...                             /* action statements */

putchar(d);                         /* character output */

printf("%3d %7.4f", k, y);          /* numerical output */
```

Input means to provide the program with some data to be used in the program and Output means to display data on screen or write the data to a printer or a file.

C programming language provides many built-in functions to read any given input and to display data on screen when there is a need to output the result.

In this tutorial, we will learn about such functions, which can be used in our program to take input from user and to output the result on screen.

All these built-in functions are present in C header files, we will also specify the name of header files in which a particular function is defined while discussing about it.

scanf() and printf() functions:

The standard input-output header file, named stdio.h contains the definition of the functions printf() and scanf(), which are used to display output on screen and to take input from user respectively.

```
#include<stdio.h>

void main()
{
```

```
    // defining a variable
int i;
    /*


displaying message on the screen
asking the user to input a value
    */
    printf("Please enter a value...");    /*
    */
reading the value entered by the user


     */
    scanf("%d", &i);
/*


    displaying the number as output
    */
    printf( "\nYou entered: %d", i);
}
```

When you will compile the above code, it will ask you to enter a value. When you will enter the value, it will display the value you have entered on screen.

You must be wondering what is the purpose of %d inside the scanf() or printf() functions. It is known as format string and this informs the scanf() function, what type of input to expect and in printf() it is used to give a heads up to the compiler, what type of output to expect.

Format String	Meaning
%d	Scan or print an integer as signed decimal number
%f	Scan or print a floating point number
%c	To scan or print a character
%s	To scan or print a character string. The scanning ends at whitespace.

We can also limit the number of digits or characters that can be input or output, by adding a number with the format string specifier, like "%1d" or "%3s", the first one means a single numeric digit and the second one means 3 characters, hence if you try to input 42, while scanf() has "%1d", it will take only 4 as input. Same is the case for output.

In C Language, computer monitor, printer etc output devices are treated as files and the same process is followed to write output to these devices as would have been followed to write the output to a file.

NOTE : printf() function returns the number of characters printed by it, and scanf() returns the number of characters read by it.

int i = printf("studytonight");

In this program printf("study tonight"); will return 12 as result, which will be stored in the variable i, because studytonight has 12 characters.

getchar() & putchar() functions:

The getchar() function reads a character from the terminal and returns it as an integer. This function reads only single character at a time. You can use this method in a loop in case you want to read more than one character. The putchar() function displays the character passed to it on the screen and returns the same character. This function too displays only a single character at a time. In case you want to display more than one characters, use putchar() method in a loop.

```c
#include <stdio.h>

void main( )
{


    int c;
    printf("Enter a character");
/*

Take a character as input and
store it in variable c

    */
    c = getchar();
    /*

        display the character stored
in variable c
    */
    putchar(c);

}
```

When you will compile the above code, it will ask you to enter a value. When you will enter the value, it will display the value you have entered.

THE gets AND puts FUNCTIONS:

C contains a number of other library functions that permit some form of data transfer into or out of the computer. the gets and puts functions, which facilitate the transfer of strings between the computer and the standard input output devices. Each of these functions accepts a single argument. The argument must be a data item that represents a string. (e.g., a character array). The string may include whitespace characters. In the case of gets, the string will be entered from the keyboard, and will

terminate with a newline character (i.e., the string will end when the user presses the Enter key).

The gets and puts functions offer simple alternatives to the use of scanf and printf for reading and displaying strings, as illustrated in the following example.

EXAMPLE - Reading and Writing a Line of Text

```
#include <stdio.h>
main( ) /* read and write a line of text */
{
char line[80];
gets(line);
puts(line);
}
```

This program utilizes gets and puts, rather than scanf and printf, to transfer the line of text into and out of the computer.

gets() & puts() functions:

The gets() function reads a line from stdin(standard input) into the buffer pointed to by str pointer, until either a terminating newline or EOF (end of file) occurs. The puts() function writes the string str and a trailing newline to stdout.

str → This is the pointer to an array of chars where the C string is stored. (Ignore if you are not able to understand this now.)

```
#include<stdio.h>
void main()
{
   /* character array of length 100 */
char str[100];
   printf("Enter a string");
   gets( str );
   puts( str );
   getch();
}
```

When you will compile the above code, it will ask you to enter a string. When you will enter the string, it will display the value you have entered.

Difference between scanf() and gets():

The main difference between these two functions is that scanf() stops reading characters when it encounters a space, but gets() reads space as character too.

If you enter name as Study Tonight using scanf() it will only read and store Study and will leave the part after space. But gets() function will read it completely

Single Character Input: the getchar Function:

The getchar function is a part of the standard C input/output library. It returns a single character from a standard input device (typically a keyboard). The function does not require any arguments, though a pair of empty parentheses must follow the word getchar.

In general, a function reference would be written as:
character variable = getchar( );

where character variable refers to some previously declared character variable.

If an end-of-file condition is encountered when reading a character with the getchar function, the value of the symbolic constant EOF will automatically be returned. (This value will be assigned within the stdio.h file. Typically, the EOF will be assigned the value -1). The detection of EOF in this manner offers a convenient way to detect an end of file, whenever and wherever it may occur. Appropriate corrective action may then be taken.

The getchar function can also be used to read multi-character strings by reading one character at a time within a multi-pass loop.

Single Character Output: the putchar Function:

Single characters can be displayed using the C library function putchar. This function is complementary to the character input function getchar.

The putchar function like getchar is a part of the standard C input/output library. It transmits a single character to the standard output device (the computer screen). The character being transmitted will be represented as a character-type variable. It must be expressed as an argument to the function, enclosed in parentheses, following the word putchar.

In general, a function reference would be written as:
putchar(character variable)
where character variable refers to some previously declared character variable.

A simple example demonstrating the use of getchar and putchar is given below:

```
#include<stdio.h>

int main()

{

char c;

printf("n Please enter a character:");

c=getchar();

printf("n The character entered is: ");

putchar(c);

return 0;

}
```

In the above program, the statement c=getchar(); accepts a character entered by the user and stores it in the variable c. The character entered by the user can be anything from the C character set. The statement putchar(c); prints the character stored in the variable c.

The putchar function can be used to output a string constant by storing the string within a one-dimensional character-type array. Each character can then be written separately within a loop. The most convenient way to do this is to utilize the for statement, which we will discuss in future.

Entering Input Data: the scanf Function:

Input data can be entered from a standard input device by means of the C library function scanf. This function can be used to enter any combination of numeric values, single characters and strings. The function returns the number of data items that have been entered successfully.

In general terms, the scanf function is written as:

scanf(control string, arg1, arg2,.   ,argN)

where **control string** refers to a string containing certain required formatting information, and **arg1,arg2,….,argN** are arguments that represent the individual data items. (Actually the arguments represent **pointers** that indicate the **addresses** of the data items within the computer's memory. We will discuss pointers in greater detail in a future article, but until then it would be helpful to remember the fact that the

arguments in the scanf function actually represent the addresses of the data items being entered.)

The control string consists of the individual group of characters called **format specifiers**, with one character group for each input data item. Each character group must begin with a per cent sign (%) and be followed by a **conversion character** which indicates the type of the data item. Within the control string, multiple character groups can be contiguous, or they can be separated by whitespace characters (ie, white spaces, tabs or newline characters).

The most commonly used conversion characters are listed below:

Conversion Character Data type of input data

c	character

d	decimal integer
e	floating point value
f	floating point value

g	floating point value

h	short integer
i	decimal, hexadecimal or octal integer

o	hexadecimal
x	integer octal integer

s	string
u	unsigned decimal integer

[. . .]	string which may include whitespace characters

The arguments to a scanf function are written as variables or arrays whose types match the corresponding character groups in the control string. Each variable name must be preceded by an **ampersand (&)**. However, character array names do not begin with an ampersand. The actual values of the arguments must correspond to the arguments in the scanf function in number, type and order.

If two or more data items are entered, they must be separated by whitespace characters. The data items may continue onto two or more lines, since the newline character is considered to be a whitespace character and can therefore separate consecutive data items.

Example 1:

```
#include<stdio.h>

int main()

{

char a[20];

int i;

float b;

printf(" n Enter the value of a, i and b");

scanf("%s %d %f", a, &i, &b);

return 0;

}
```

In the above program, within the scanf function, the **control string** is "%s %d %f". It denotes three-character groups or format specifiers. The first format specifier %s denotes that the first argument a represents a string (character array), the second format specifier %d denotes that the second argument i is an integer and the third format specifier %f denotes that the third argument b is a floating point number.

Also note that the only argument not preceded by an ampersand (&) is a since a denotes a character array.

The **s-type** conversion character applies to **a string that is terminated by a whitespace character**. Therefore a string that **includes whitespace characters** cannot be entered in this manner. To do so, there are two ways:

1. The s-type conversion character is replaced by a sequence of characters enclosed in square brackets, designated as [. . .]. Whitespace characters are also included in the string so that a string that contains whitespaces may be read.

`When the program is executed, successive characters will continue to be read as long as each input character matches one of the characters enclosed in the square brackets. The order of the characters in the square brackets need not correspond to the order of the characters being entered. As soon as an input character is encountered that does not match one of the characters within the brackets, the scanf function will

stop reading any more characters and will terminate the string. A null character will then automatically be added to the end of the string.

Example:
#include<stdio.h>

int main()

{

char line[80];

scanf(" %[ABCDEFGHIJKLMNOPQRSTUVWXYZ]", line);

printf("%s", line);

return 0;

}

If the input data is:
READING A STRING WITH WHITE SPACES

then the entire data will be assigned to the array line. However, if the input is:

Reading A String With white Spaces

then only the letters in uppercase (R, A, S, W, S) will be assigned to line, as all the characters in the control string are in uppercase.

2. To enter a string that includes whitespaces as well as uppercase and lowercase characters we can use the **circumflex, ie (^)**, followed by a newline character within the brackets.

Example:
scanf("[^n]", line);

The circumflex causes the subsequent characters within the square brackets to be interpreted in the opposite manner. Thus, when the program is executed, characters will be read as long as a newline character is not encountered.

Reading Numbers: Specifying Field Width:

The consecutive non-whitespace characters that define a data item collectively define a **field**. To limit the number of such characters for a data item, an unsigned integer indicating the **field width** precedes the

conversion character. The input data may contain fewer characters than the specified field width. Extra characters will be ignored.

Example: If a and b are two integer variables and the following statement is being used to read their values:

scanf("%3d %3d", &a, &b);

and if the input data is: 1 4

then a will be assigned 1 and b 4.
If the input data is 123 456 then a=123 and b=456.

If the input is 1234567, then a=123 and b=456. 7 is ignored.

If the input is 123 4 56 (space inserted by a typing mistake), then a=123 and b=4. This is because the space acts as a data item separator.

Example 1: C Output:

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

// Displays the string inside quotations
printf("C Programming");
return 0;


}
```

Output:
C Programming

Example 2: Integer Output

```
#include <stdio.h>
int main()
{
int testInteger = 5;
printf("Number = %d", testInteger);
return 0;
}
```

Example 2: Integer Output:

```
#include <stdio.h>
int main()
{
int testInteger = 5;
```

```
printf("Number = %d", testInteger);
return 0;
}
```

Output
Number = 5
Example 3: float and double Output

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

float number1 = 13.5;
double number2 = 12.4;
printf("number1 = %f\n", number1);
printf("number2 = %lf", number2);

return 0;
}
```

Output
number1 = 13.500000
number2 = 12.400000

Example 4: Print Characters:
```
#include <stdio.h>
int main()
{

char chr = 'a';
printf("character = %c.", chr);
return 0;

}
```

Output
character = a

Example 5: Integer Input/Output:
```
#include <stdio.h>
int main()
{
int testInteger;
printf("Enter an integer: ");
scanf("%d", &testInteger);
printf("Number = %d",testInteger);
return 0;
}
```

Output
Enter an integer: 4
Number = 4

Example 6: Float and Double Input/Output:

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

float num1;
double num2;

printf("Enter a number: ");
scanf("%f", &num1);
printf("Enter another number: ");
scanf("%lf", &num2);

printf("num1 = %f\n", num1);
printf("num2 = %lf", num2);

return 0;
}
```

Output
Enter a number: 12.523
Enter another number: 10.2
num1 = 12.523000
num2 = 10.200000

Example 7: C Character I/O:

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

char chr;
printf("Enter a character: ");
scanf("%c",&chr);
printf("You entered %c.", chr);
return 0;

}
```

Output
Enter a character: g
You entered g.

Example 8: ASCII Value:

```c
#include <stdio.h>
int main()
{
char chr;
printf("Enter a character: ");
scanf("%c", &chr);

// When %c is used, a character is displayed
printf("You entered %c.\n",chr);

// When %d is used, ASCII value is displayed
printf("ASCII value is % d.", chr);
return 0;

}
```

Output
Enter a character: g
You entered g.
ASCII value is 103.

Example 9 : I/O Multiple Values:

```c
#include <stdio.h>
int main()
{
int a;
float b;

printf("Enter integer and then a float: ");

// Taking multiple inputs
scanf("%d%f", &a, &b);

printf("You entered %d and %f", a, b);
return 0;
}
```

Output
Enter integer and then a float: -3
3.4
You entered -3 and 3.400000

Here's a list of commonly used C data types and their format specifiers.

Data Type	Format Specifier
int	%d
char	%c
float	%f

double	%lf
short int	%hd
unsigned int	%u
long int	%li
long long int	%lli
unsigned long int	%lu
unsigned long long int	%llu
signed char	%c
unsigned char	%c
long double	%Lf

Interactive (Conversational) Programming:

Many modern computer programs are designed to create an interactive dialog between the computer and the person using the program (the "user"). These dialogs usually involve some form of question-answer interaction, where the computer asks the questions and the user provides the answers, or vice versa. The computer and the user thus appear to be carrying on some limited form of conversation.

In C, such dialogs can be created by alternate use of the scanf and printf functions. The actual programming is straightforward, though sometimes confusing to beginners, since the printf function is used both when entering data (to create the computer's questions) and when displaying results. On the other hand, scanf is used only for actual data entry.

The basic ideas are illustrated in the following example.

EXAMPLE - Averaging Student Exam Scores This example presents a simple, interactive C program that reads in a student's name and three exam scores, and then calculates an average score. The data will be entered interactively, with the computer asking the user for information and the user supplying the information in a free format, as requested. Each input data item will be entered on a separate line. Once all of the data have been entered, the computer will compute the desired average and write out all of the data (both the input data and the calculated average).

The actual program is shown below.

```
#include <stdio.h>
main( ) /* sample interactive program */
{
char name[20];
float score1 , score2, score3, avg;
printf("P1ease enter your name: "); /* enter name */
scanf(" %["\n]", name);
printf("P1ease enter the first score: "); /* enter 1st score */
scanf("%f", &score1);
```

```
printf("Please enter the second score: "); /* enter 2nd score */
scanf ( "%f",&score2) ;
printf("P1ease enter the third score: "); /* enter 3rd score */
scanf("%f", &score3);
avg = (score1+score2+score3)/3; /* calculate avg */
printf ( "\n\nName: %-s\n\n", name); /* write output */
printf("Score 1: %-5.1f\n", score1);
printf("Score 2: %-5.1f\n", score2);
printf("Score 3: %-5.1f\n\n", score3);
printf("Average: %-5.1f\n\n", avg);
}
```

Notice that two statements are associated with each input data item. The first is a printf statement, which generates a request for the item. The second statement, a scanf function, causes the data item to be entered from the standard input device (i.e., the keyboard).

After the student's name and all three exam scores have been entered, an average exam score is calculated. The input data and the calculated average are then displayed, as a result of the group of printf statements at the end of the program.

A typical interactive session is shown below. To illustrate the nature of the dialog, the user's responses have been underlined.

```
Please enter your name        : Robert Smith
Please enter the first score      88
Please enter the second score : 62.3
Please enter the third score    :
Name                          : Robert Smith
Score 1                        : 88.0
Score 2                        : 62.5
Score 3                        : 90.0
Average                        : 80.2
```

5.2 UNIT END QUESTIONS

1. Explain scanf() and printf() functions in detail.
2. Explain getchar() & putchar() functions in detail.
3. Write a Difference between scanf() and gets()
4. Enlist most commonly used conversion characters in C
5. What is Format specifiers ? Explain with Example.
6. What is INTERACTIVE (CONVERSATIONAL) PROGRAMMING? Explain

*****

UNIT II

6

CONDITIONAL STATEMENTS

Unit Structure

6.0   Objectives

6.1   Introduction

6.2   Unit End Questions

6.0 OBJECTIVES

In most of the C programs we have encountered so far, the instructions were executed in the same order in which they appeared within the program. Each instruction was executed once and only once.

Programs of this type are unrealistically simple, since they do not include any logical control structures.

Thus, these programs did not include tests to determine if certain conditions are true or false, they did not require the repeated execution of groups of statements, and they did not involve the execution of individual groups of statements on a selective basis.

Most C programs that are of practical interest make extensive use of features such as these.

For example, a realistic C program may require that a logical test be carried out at some particular point within the program. One of several possible actions will then be carried out, depending on the outcome of the logical test. This is known as branching. There is also a special kind of branching, called selection, in which one group of statements is selected from several available groups. In addition, the program may require that a group of instructions be executed repeatedly, until some logical condition has been satisfied. This is known as looping. Sometimes the required number of repetitions is known in advance; and sometimes the computation continues indefinitely until the logical condition becomes true.

All of these operations can be carried out using the various control statements included in C. We will see how this is accomplished in this chapter. The use of these statements will open the door to programming problems that are much broader and more interesting than those considered earlier.

6.1 INTRODUCTION

Decision Making:

Decision making structures require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Show below is the general form of a typical decision making structure found in most of the programming languages –
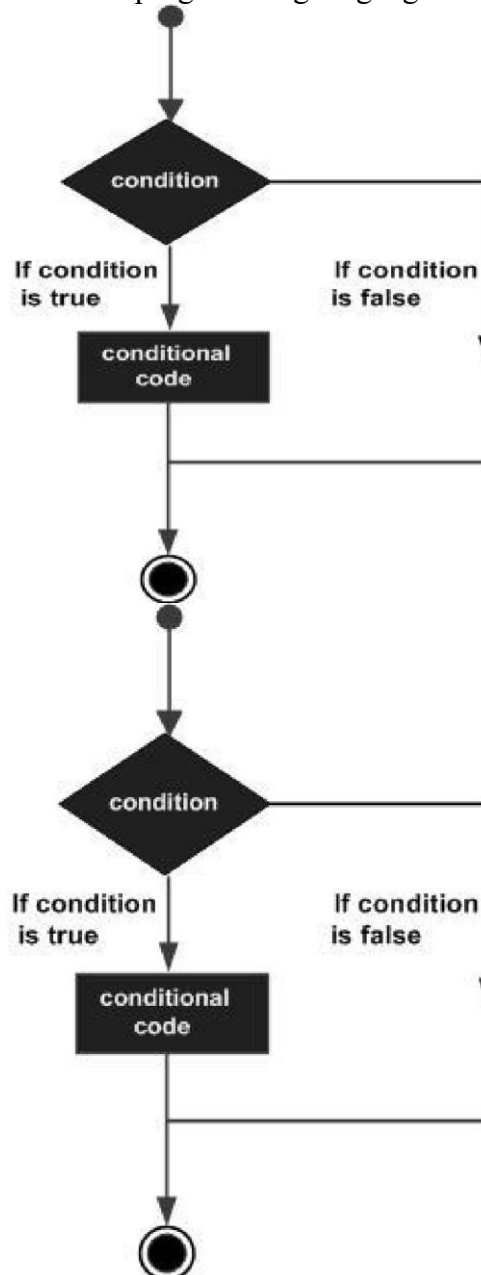
C programming language assumes any **non-zero** and **non-null** values as **true**, and if it is either **zero** or **null**, then it is assumed as **false** value.

C programming language provides the following types of decision making statements.

Sr.No.	Statement & Description
1	**if statement** An **if statement** consists of a boolean expression followed by one or more statements.
2	**if...else statement** An **if statement** can be followed by an optional **else statement**, which executes when the Boolean expression is false.
4	**switch statement** A **switch** statement allows a variable to be tested for equality against a list of values.
5	**nested switch statements** You can use one **switch** statement inside another **switch** statement(s).

The ? : Operator

We have covered **conditional operator ? :** in the previous chapter which can be used to replace **if...else** statements. It has the following general form –

Exp1 ? Exp2 : Exp3;

Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

The value of a ? expression is determined like this −

- Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ? expression.
- If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

Decision making in C:

Decision making is about deciding the order of execution of statements based on certain conditions or repeat a group of statements until certain specified conditions are met. C language handles decision-making by supporting the following statements,

- if statement
- switch statement
- conditional operator statement (? : operator)
- goto statement

Decision making with if statement:

The if statement may be implemented in different forms depending on the complexity of conditions to be tested. The different forms are,

1. Simple if statement
2. if. ...else statement
3. Nested if. ...else statement
4. Using else if statement

The following table shows all the relational operators supported by C. Assume variable **A** holds 20 and variable **B** holds 30 then –

Relational Operators:

Operator	Description	Example
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

Example

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

int a = 19;
int b = 9;
int c ;

if( a == b ) {
   printf("a is equal to b \n" );
} else {
   printf("a is not equal to b \n" );
}
```

```
if ( a < b ) {
printf("a is less than b \n" );
} else {
   printf("a is not less than b \n" );
}

if ( a > b ) {
   printf("a is greater than b \n" );
} else {
   printf("a is not greater than b \n" );
}

/* Lets change value of a and b */
a = 5;
b = 20;

if ( a <= b ) {
   printf("a is either less than or equal to b\n" );
}

if ( b >= a ) {
   printf("b is either greater than or equal to b\n" );
}
}
```

Result:
a is not equal to b
a is not less than b
a is greater than b
a is either less than or equal to b
b is either greater than or equal to b

Logical Connectives:

C contains two logical connectives (also called logical operators), && (AND) and I I (OR), and the unary negation operator ! . The logical connectives are used to combine logical expressions, thus forming more complex expressions. The negation operator is used to reverse the meaning of a logical expression (e.g., from true to false).

EXAMPLE 6.2:
Here are some logical expressions that illustrate the use of the logical connectives and the negation
operator.
(count <= 100) && (ch1 != ' * ')
(balance < 1000.0) || (status == 'R')
(answer < 0) || ((answer > 5.0) && (answer <= 10.0))
! ((pay >= 1000.0) && (status == 's'))

92

Note that ch1 and status are assumed to be char-type variables in these examples. The remaining variables are assumed to be numeric (either integer or floating-point). Since the relational and equality operators have a higher precedence than the logical operators, some of the parentheses are not needed in the above expressions. Thus, we could have written these expressions as

count <= 100 && ch1 != '*'

balance < 1000.0 || status == 'R'

answer < 0 || answer > 5.0 && answer <= 10.0

! (pay >= 1000.0 && status == 's')

It is a good idea, however, to include pairs of parentheses if there is any doubt about the operator precedences. This is particularly true of expressions that are relatively complicated, such as the third expression above.

Following table shows all the logical operators supported by C language. Assume variable **A** holds 1 and variable **B** holds 0, then

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

Example:

```
#include <stdio.h>
main() {
int a = 5;
int b = 20;
int c ;
if ( a && b ) {
printf("Condition is true\n" );
}
if ( a || b ) {
printf("Condition is true\n" );
}
/* lets change the value of a and b */
a = 0;
```

93

```
b = 10;
if ( a && b ) {
printf("Condition is true\n" );
} else {
printf("Condition is not true\n" );
}
if ( !(a && b) ) {
printf("Condition is true\n" );
}
}
```

Result:
Condition is true
Condition is true
Condition is not true
Condition is true

Simple if statement:

if statement is used for branching when a single condition is to be checked. The condition enclosed in if statement decides the sequence of execution of instruction. If the condition is true, the statements inside if statement are executed, otherwise they are skipped. In C programming language, any non zero value is considered as true and zero or null is considered false.

Syntax of if statement

if (condition)

{

statements;
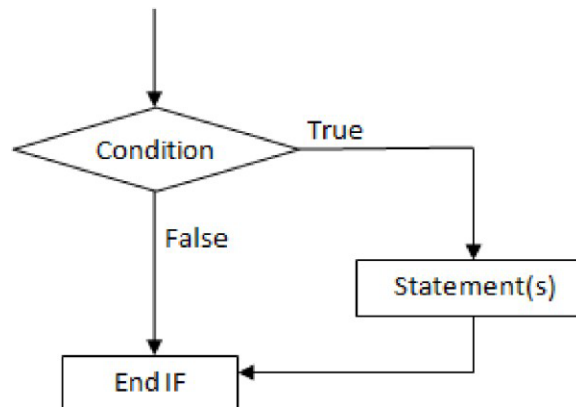... ... ...

}

Example 1 :
```
#include <stdio.h>
void main( )
{
int x, y;
x = 15;
y = 13;
if (x > y )
{
    printf("x is greater than y");
}
}
```

OUTPUT:
x is greater than y

Flowchart:

Example 2 :- C program to print the square of a number if it is less than 10.

```c
#include<stdio.h>
int main()
{
int n;
    printf("Enter a number:");
scanf("%d",&n);
if(n<10)
{
printf("%d is less than 10\n",n);
printf("Square = %d\n",n*n);
}
return 0;
}
```

This program is an example of using if statement. A number is asked from user and stored in variable *n*. If the value of *n* is less than 10, then its square is printed on the screen. If the condition is false the program, execution is terminated.

Output
Enter a number:6
6 is less than 10
Square = 36

if ... else statement :
if ... else statement is a two way branching statement. It consists of two blocks of statements each enclosed inside if block and else block respectively. If the condition inside if statement is true, statements inside

95

if block are executed, otherwise statements inside else block are executed.
Else block is optional and it may be absent in a program.

Syntax of if...else statement
if (condition)
{
statements;
... ...
}
else
{
statements;
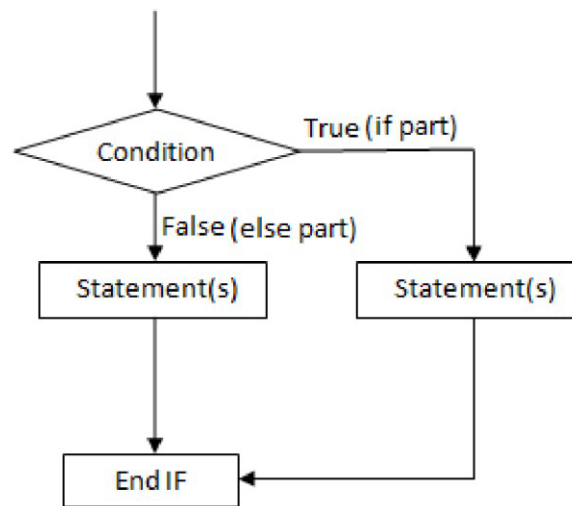... ... ...
}

Flowchart of if ... else statement

fig: Flowchart for if ... else statement

Example of if ... else statement

Example 2: C program to find if a number is odd or even.

```
#include<stdio.h>
int main()
{
int n;
printf("Enter a number:");
scanf("%d",&n);
if(n%2 == 0)
printf("%d is even",n);
else
printf("%d is odd",n);
return 0;
}
```

Here, a number is entered by user which is stored in n. The if statement checks if the remainder of that number when divided by 2 is zero or not. If the remainder is zero, the number is even which is printed on the screen. If the remainder is 1, the number is odd.

Note: If there is only one statement inside if block, we don't need to enclose it with curly brackets { }.

Output :

```
Enter a number:18

18 is even

Enter a number:33

33 is odd
```

if ... else if ... else statement:

It is used when more than one condition is to be checked. A block of statement is enclosed inside if, else if and else part. Conditions are checked in each if and else if part. If the condition is true, the statements inside that block are executed. If none of the conditions are true, the statements inside else block are executed. A if ... else if ... else statement must have only one if block but can have as many else if block as required. Else part is optional and may be present or absent.

Syntax of if...else if...else statement

```
if (condition 1)
{
statements;
... ... ...
}
else if (condition 2)
{
statements;
... ... ...
}
... ... ...
... ... ...
else if (condition n)
{
statements;
... ... ...
}
else
```

```
 {
statements;
... ... ...
}
```

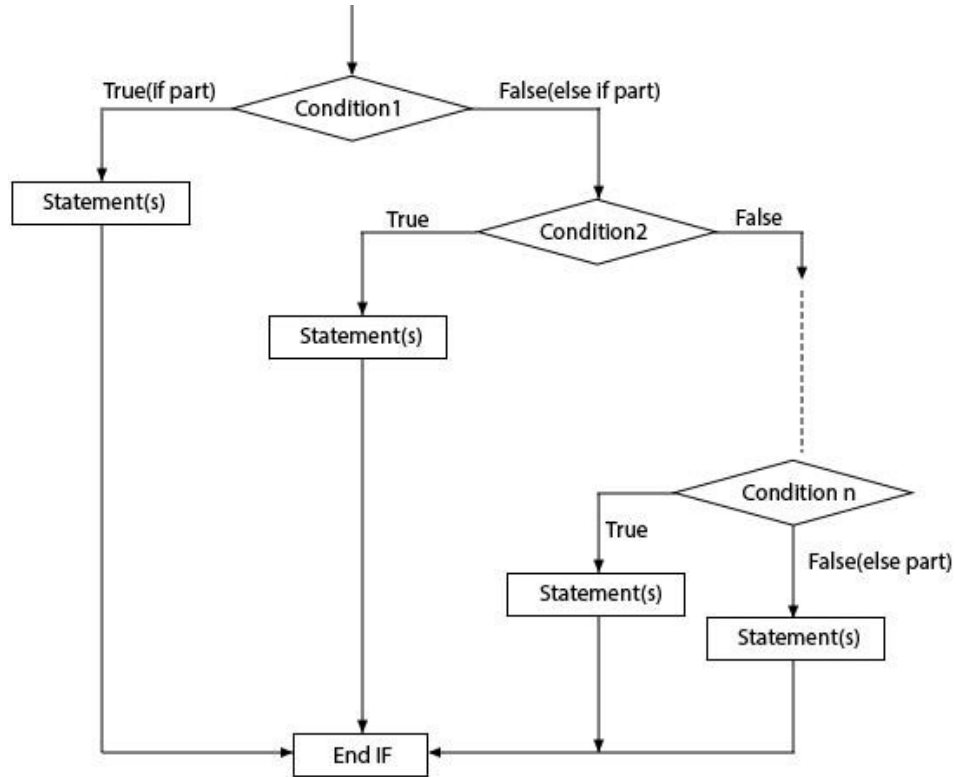**Flowchart of if ... else if ... else statement**

fig: Flowchart for if ... else if ... else statement

Example of if ... else if ... else statement

Example 3: C program to find if a number is negative, positive or zero.

```
#include<stdio.h>
int main()
{
int n;
printf("Enter a number:");
scanf("%d",&n);
if(n<0)
printf("Number is negative");
else if(n>0)
printf("Number is positive");
else
printf("Number is equal to zero");
return 0;
}
```

In this program, a number is entered by user stored in variable *n*. The if ... else if ... else statement tests two conditions:

1. *n<0*: If it is true, "Number is negative" is printed on the screen.

2. *n>0*: If it is true, "Number is positive" is printed on the screen.

If both of these conditions are false then the number is zero. So the program will print "Number is zero".

Output

Enter a number:109

Number is positive

Enter a number:-56

Number is negative

Enter a number:0

Number is equal to zero

6.2 UNIT END QUESTIONS

1. Explain switch case with Example.

2. What is the difference Between Switch case and If-else Condition ?

3. Draw Flowchart for if ... else if ... else statement.

4. Write a C program to find if a number is negative, positive or zero.

*****

7

LOOPS

Unit Structure

7.0   Objectives

7.1   Introduction

7.2   Unit End Questions

7.0 OBJECTIVES

A loop is used for executing a block of statements repeatedly until a given condition returns false. In the previous tutorial we learned for loop. In this guide we will learn while loop in C.

C – while loop:
Syntax of while loop:

```
while (condition test)
{
//Statements to be executed repeatedly
// Increment (++) or Decrement (--) Operation
}
Flow Diagram of while loop
C while loop
```

Example of while loo:

```
#include <stdio.h>
int main()
{
int count=1;
while (count <= 4)
{
printf("%d ", count);
count++;
}
return 0;
}
Output: 1 2 3 4
```

Step1: The variable count is initialized with value 1 and then it has been tested for the condition.

step2: If the condition returns true then the statements inside the body of while loop are executed else control comes out of the loop.

100

step3: The value of count is incremented using ++ operator then it has been tested again for the loop condition.

Guess the output of this while loop ?

```c
#include <stdio.h>
int main()
{
int var=1;
while (var <=2)
{
printf("%d ", var);
}
}
```

The program is an example of infinite while loop. Since the value of the variable var is same (there is no ++ or – operator used on this variable, inside the body of loop) the condition var<=2 will be true forever and the loop would never terminate.

Examples of infinite while loop:
Example :

```c
#include <stdio.h>

int main()
{
int var = 6;
while (var >=5)
{
printf("%d", var);
var++;
}
return 0;
}
```

Infinite loop: var will always have value >=5 so the loop would never end.

do while loop in C:

The do while loop is a post tested loop. Using the do-while loop, we can repeat the execution of several parts of the statements. The do-while loop is mainly used in the case where we need to execute the loop at least once. The do-while loop is mostly used in menu-driven programs where the termination condition depends upon the end user.

do while loop syntax:
The syntax of the C language do-while loop is given below:

101

```
do{
//code to be executed
}while(condition);
```

Example:

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

char c;
int choice,dummy;
do{
printf("\n1. Print Hello\n2. Print demo program \n3. Exit\n");
scanf("%d",&choice);
switch(choice)
{
case 1 :
printf("Hello");
break;
case 2:
printf("demo program");
break;
case 3:
exit(0);
break;
default:
printf("please enter valid choice");

}
printf("do you want to enter more?");
scanf("%d",&dummy);
scanf("%c",&c);
}while(c=='y');
}
```

Output
Print Hello
Print Demo program
Exit
1
Hello
do you want to enter more?
y
Print Hello
Print demo program
Exit
2
Demo program

102

do you want to enter more?
N

Flowchart of do while loop
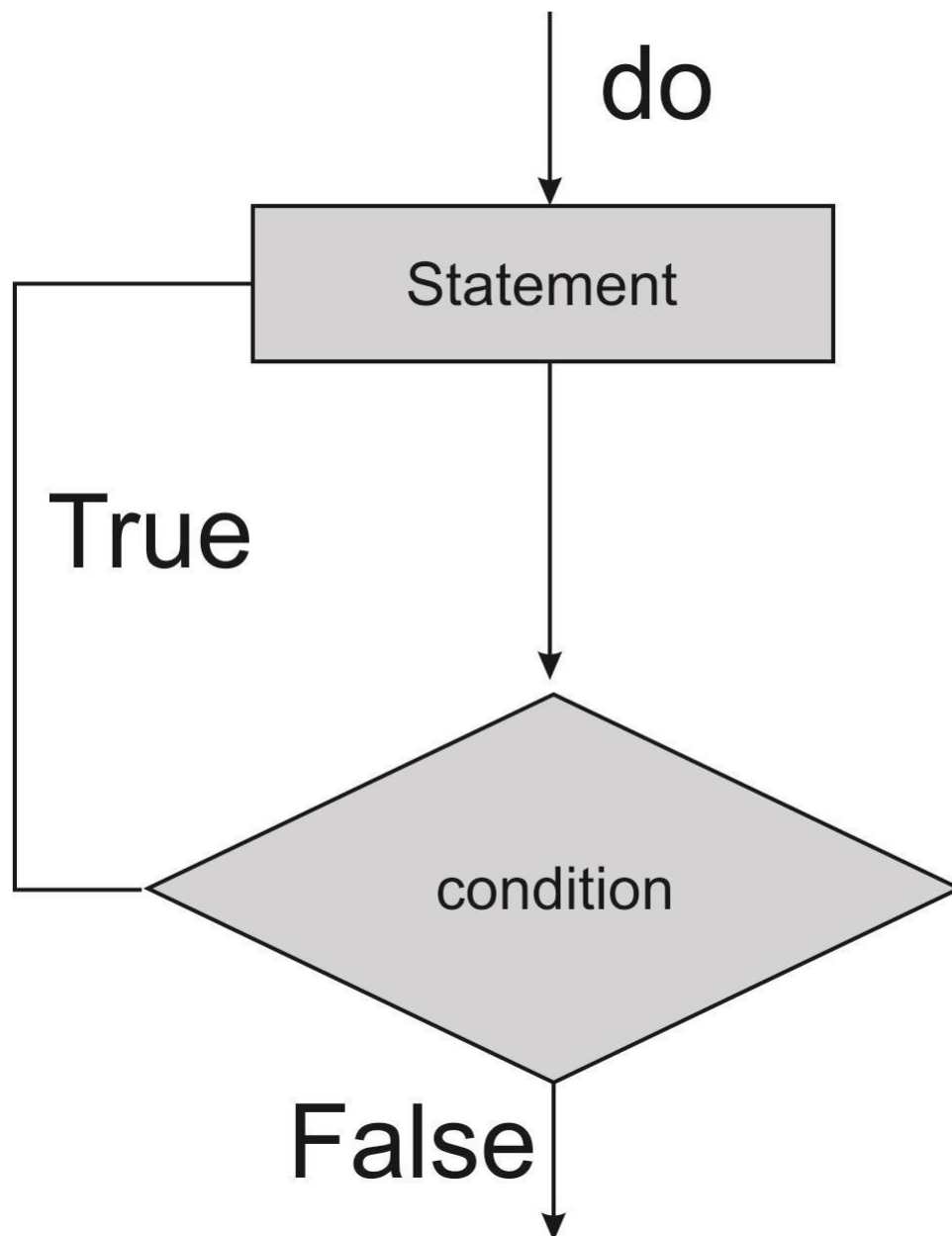
Program :-

```
#include<stdio.h>
int main(){
int i=1,number=0;
printf("Enter a number: ");
scanf("%d",&number);
do{
```

```
printf("%d \n",(number*i));
i++;
}while(i<=10);
return 0;
}
```

Output

Enter a number: 5

5

10

15

20

25

30

35

40

45

50

Enter a number: 10

10

20

30

40

50

60

70

80

90

100

for loop in C:

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax:

The syntax of a **for** loop in C programming language is −

for (init; condition; increment)
{
statement(s);
}

Here is the flow of control in a 'for' loop −

- The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.

- Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the 'for' loop.

- After the body of the 'for' loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.

- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the 'for' loop terminates.

Flowchart :


```
for( init; condition; increment )
{
    conditional code ;
}
```

Example

105
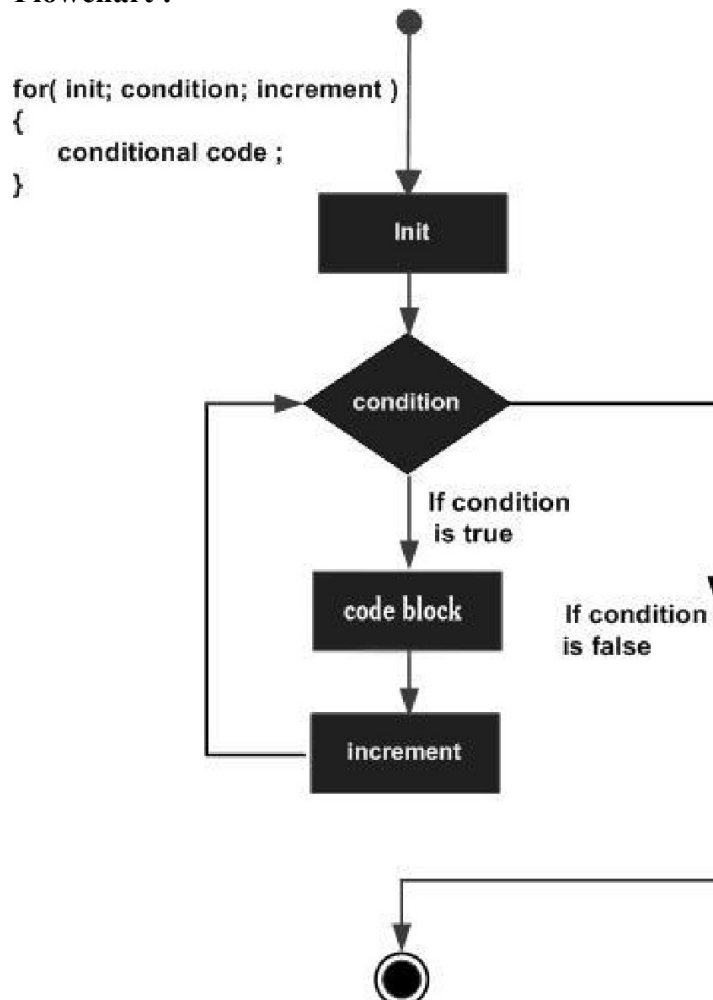
```c
#include <stdio.h>
int main () {
int a;
/* for loop execution */
for( a = 10; a < 20; a = a + 1 ){
printf("value of a: %d\n", a);
}
return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

The Infinite Loop:

A loop becomes an infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the 'for' loop are required, you can make an endless loop by leaving the conditional expression empty.

```c
#include <stdio.h>
int main () {
for( ; ; ) {
printf("This loop will run forever.\n");
}
return 0;
}
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the for(;;) construct to signify an infinite loop.

NOTE − You can terminate an infinite loop by pressing Ctrl + C keys.

nested loops in C:

C programming allows to use one loop inside another loop. The following section shows a few examples to illustrate the concept.

Syntax

The syntax for a nested for loop statement in C is as follows –

```
for ( init; condition; increment ) {
for ( init; condition; increment ) {
statement(s);
}
statement(s);
}
```

Example:
The following program uses a nested for loop to find the prime numbers from 2 to 100 –

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

/* local variable definition */
int i, j;
for(i = 2; i<100; i++) {

for(j = 2; j <= (i/j); j++)
if(!(i%j)) break; // if factor found, not prime
if(j > (i/j)) printf("%d is prime\n", i);
}
return 0;
}
```

Result :
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime

37 is prime

41 is prime

43 is prime

47 is prime

53 is prime

59 is prime

61 is prime

67 is prime

71 is prime

73 is prime

79 is prime

83 is prime

89 is prime

97 is prime

C switch Statement: The switch statement allows us to execute one code block among many alternatives.

You can do the same thing with the if...else..if ladder. However, the syntax of the switch statement is much easier to read and write.

Syntax of switch...case
```
switch (expression)
{
case constant1:
// statements
break;
case constant2:
// statements
break;
.
.
.
default:
// default statements
}
```

How does the switch statement work?:

The expression is evaluated once and compared with the values of each case label.

If there is a match, the corresponding statements after the matching label are executed. For example, if the value of the expression is equal to constant2, statements after case constant2: are executed until break is encountered.

If there is no match, the default statements are executed.

If we do not use break, all statements after the matching label are executed. the default clause inside the switch statement is optional.
switch Statement Flowchart
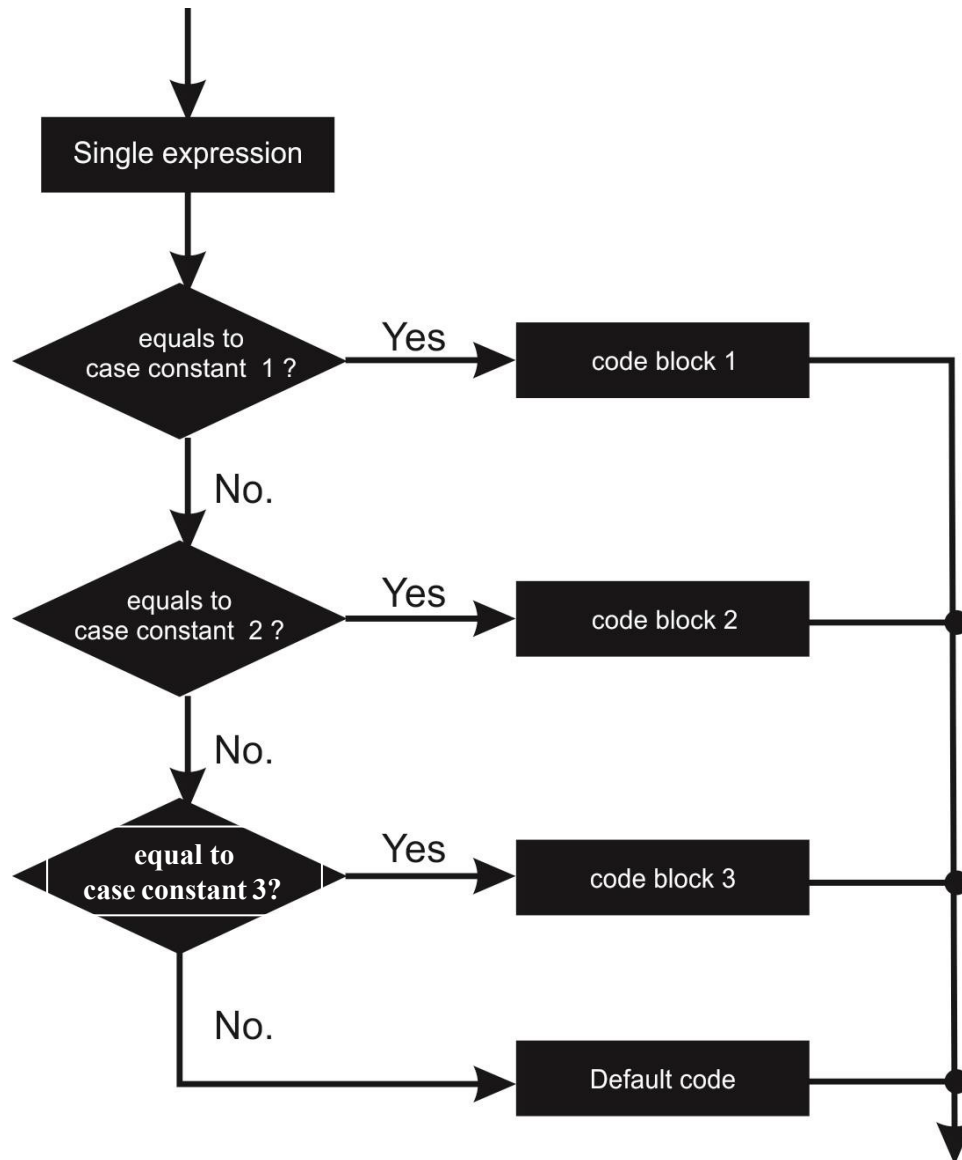
Example:

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

int i=2;
switch (i)
{
case 1:
printf("Case1 ");
```

```
break;
case 2:
printf("Case2 ");
break;
case 3:
printf("Case3 ");
break;
case 4:
printf("Case4 ");
break;
default:
printf("Default ");
}
return 0;
}
```

Output:
Case 2

7.3 UNIT END QUESTIONS

1. Explain While Loop with Example.

2. How to create infinite while loop ? Give any example

3. Give and Explain Syntax for Do-While Loop

4. Draw FlowChart for For loop. Also discuss syntax for same.

5. How does the switch statement work? Explain with Flowchart.

6. Write a C program to print prime numbers between 1 to 100 using nested for loop.

*****

8

FUNCTIONS

Unit Structure

8.0   Objectives

8.1   Introduction

8.2   Unit End Questions

8.0 OBJECTIVES

Function is a group of statements that together perform a task. Every C program has at least one function, which is main(), and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is so each function performs a specific task.

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, function strcat() to concatenate two strings, function memcpy() to copy one memory location to another location and many more functions.

A function is known with various names like a method or a sub-routine or a procedure, etc.

Defining a Function:

The general form of a function definition in C programming language is as follows:
return_type function_name(parameter list)
{
body of the function
}

A function definition in C programming language consists of a function header and a function body. Here are all the parts of a function:

- **Return Type:** A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.

111

- **Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the function signature.

- **Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

- **Function Body:** The function body contains a collection of statements that define what the function does.

Example:
Following is the source code for a function called max(). This function takes two parameters num1 and num2 and returns the maximum between the two:

```
/* function returning the max between two numbers */
int max(int num1, int num2)
{
/* local variable declaration */
int result;
if (num1 > num2)
result = num1;
else
result = num2;
return result;
}
```

Function Declarations:

A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts:

return_type function_name(parameter list);

For the above defined function max(), following is the function declaration:
int max(int num1, int num2);

Parameter names are not important in function declaration only their type is required, so following is also valid declaration:
int max(int, int);

Function declaration is required when you define a function in one source file and you call that function in another file. In such case you should declare the function at the top of the file calling the function.

Calling a Function:

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, program control is transferred to the called function. A called function performs defined task, and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example:

```c
#include <stdio.h>
/* function declaration */
int max(int num1, int num2);
int main ()
{
/* local variable definition */
int a = 100;
int b = 200;
int ret;
/* calling a function to get max value */
ret = max(a, b);
printf( "Max value is : %d\n", ret );
return 0;
}
/* function returning the max between two numbers */
int max(int num1, int num2)
{
/* local variable declaration */
int result;
if (num1 > num2)
result = num1;
else
result = num2;
return result;
}
```

Here max() function along with main() function and compiled the source code. While running final executable, it would produce the following result:
Max value is : 200

Function Arguments:

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal

parameters of the function. The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function:

Function call by value:

The call by value method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument. By default, C programming language uses call by value method to pass arguments.

In general, this means that code within a function cannot alter the arguments used to call the function. Consider the function swap() definition as follows.

```
/* function definition to swap the values */
void swap(int x, int y)
{
int temp;
temp = x; /* save the value of x */
x = y; /* put y into x */
y = temp; /* put x into y */
return;
}
```

Now, let us call the function swap() by passing actual values as in the following example:

```
#include <stdio.h>
/* function declaration */
void swap(int x, int y);
int main ()
{
/* local variable definition */
int a = 100;
int b = 200;
printf("Before swap, value of a : %d\n", a );
printf("Before swap, value of b : %d\n", b );
/* calling a function to swap the values */
swap(a, b);
printf("After swap, value of a : %d\n", a );
printf("After swap, value of b : %d\n", b );
```

```
return 0;
}
```

Let us put above code in a single C file, compile and execute it, it will produce the following

Result:

Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200

Which shows that there is no change in the values though they had been changed inside the function.

Function call by reference:

The call by reference method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

To pass the value by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function swap(), which exchanges the values of the two integer variables pointed to by its arguments.

```
/* function definition to swap the values */
void swap(int *x, int *y)
{
int temp;
temp = *x; /* save the value at address x */
*x = *y; /* put y into x */
*y = temp; /* put x into y */
return;
}
```

Let us call the function swap() by passing values by reference as in the following example:

```
#include <stdio.h>
/* function declaration */
void swap(int *x, int *y);
int main ()
```

```
{
/* local variable definition */
int a = 100;
int b = 200;
printf("Before swap, value of a : %d\n", a );
printf("Before swap, value of b : %d\n", b );
/* calling a function to swap the values.
* &a indicates pointer to a ie. address of variable a and
* &b indicates pointer to b ie. address of variable b.
*/
swap(&a, &b);
printf("After swap, value of a : %d\n", a );
printf("After swap, value of b : %d\n", b );
return 0;
}
```

Let us put above code in a single C file, compile and execute it, it will produce the following

Result:
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200

Which shows that there is no change in the values though they had been changed inside the function.

Passing Arguments to the main Function:

The function called at program startup is named main . The main function can be defined with no parameters or with two parameters (for passing command-line arguments to a program when it begins executing). The two parameters are referred to here as *argc* and *argv*, though any names can be used because they are local to the function in which they are declared. A main function has the following syntax:

int main(void) { . . . }
int main(int *argc*, char **argv[ ]) { . . . })
argc
The number of arguments in the command line that invoked the program. The value of *argc* is nonnegative.

argv
Pointer to an array of character strings that contain the arguments, one per string. The value argv[*argc*] is a null pointer.

If the value of *argc* is greater than zero, the array members *argv*[0] through *argv*[*argc* - 1] inclusive contain pointers to strings, which are given implementation-defined values by the host environment before program startup. The intent is to supply the program with information determined before program startup from elsewhere in the host environment. If the host environment cannot supply strings with letters in both uppercase and lowercase, the host environment ensures that the strings are received in lowercase.

If the value of *argc* is greater than zero, the string pointed to by *argv*[0] represents the program name; *argv*[0][0] is the null character if the program name is not available from the host environment. If the value of *argc* is greater than one, the strings pointed to by *argv*[1] through *argv*[*argc* - 1] represent the program parameters.

The parameters *argc* and *argv*, and the strings pointed to by the *argv* array, can be modified by the program and keep their last-stored values between program startup and program termination.

In the main function definition, parameters are optional. However, only the parameters that are defined can be accessed.

Function Prototype:

A function prototype is a function declaration that specifies the data types of its arguments in the parameter list. The compiler uses the information in a function prototype to ensure that the corresponding function definition and all corresponding function declarations and calls within the scope of the prototype contain the correct number of arguments or parameters, and that each argument or parameter is of the correct data type.

Prototypes are syntactically distinguished from the old style of function declaration. The two styles can be mixed for any single function, but this is not recommended. The following is a comparison of the old and the prototype styles of declaration:

Old style:
- Functions can be declared implicitly by their appearance in a call.
- Arguments to functions undergo the default conversions before the call.
- The number and type of arguments are not checked.

Prototype style:
- Functions are declared explicitly with a prototype before they are called. Multiple declarations must be compatible; parameter types must agree exactly.

- Arguments to functions are converted to the declared types of the parameters.
- The number and type of arguments are checked against the prototype and must agree with or be convertible to the declared types. Empty parameter lists are designated using the void keyword.
- Ellipses are used in the parameter list of a prototype to indicate that a variable number of parameters are expected.

Prototype Syntax:
A function prototype has the following syntax:

function-prototype-declaration:
declaration-specifiers(opt) declarator;

The *declarator* includes a parameter type list, which can consist of a single parameter of type void . In its simplest form, a function prototype declaration might have the following format:

storage_class(opt) return_type(opt) function_name (
type(1) parameter(1), ..., type(n) parameter(n));

Consider the following function definition:
char function_name(int lower, int *upper, char (*func)(), double y)
{ }

The corresponding prototype declaration for this function is:
char function_name(int lower, int *upper, char (*func)(), double y);

A prototype is identical to the header of its corresponding function definition specified in the prototype style, with the addition of a terminating semicolon (;) or comma (,), as appropriate (depending on whether the prototype is declared alone or in a multiple declaration).

Function prototypes need not use the same parameter identifiers as in the corresponding function definition because identifiers in a prototype have scope only within the identifier list. Moreover, the identifiers themselves need not be specified in the prototype declaration; only the types are required.

For example, the following prototype declarations are equivalent:

char function_name(int lower, int *upper, char (*func)(), double y);
char function_name(int a, int *b, char (*c)(), double d);
char function_name(int, int *, char (*)(), double);

Though not required, identifiers should be included in prototypes to improve program clarity and increase the type-checking capability of the compiler.

Variable-length argument lists are specified in function prototypes with ellipses. At least one parameter must precede the ellipses. For example:

char function_name(int lower, ...);

Data-type specifications cannot be omitted from a function prototype.

Explanation:

It is now considered good form to use **function prototypes** for all functions in your program. A prototype declares the function name, its parameters, and its return type to the rest of the program prior to the function's actual declaration. To understand why function prototypes are useful, enter the following code and run it:

```
#include <stdio.h>
void main()
{
printf("%d\n",add(3));
}
int add(int i, int j)
{
return i+j;
}
```

This code compiles on many compilers without giving you a warning, even though **add** expects two parameters but receives only one. It works because many C compilers do not check for parameter matching either in type or count. You can waste an enormous amount of time debugging code in which you are simply passing one too many or too few parameters by mistake. The above code compiles properly, but it produces the wrong answer.

To solve this problem, C lets you place function prototypes at the beginning of (actually, anywhere in) a program. If you do so, C checks the types and counts of all parameter lists. Try compiling the following:

```
#include <stdio.h>
int add (int,int); /* function prototype for add */
void main()
{
printf("%d\n",add(3));
}
int add(int i, int j)
{
return i+j;
}
```

The prototype causes the compiler to flag an error on the **printf** statement. Place one prototype for each function at the beginning of your program. They can save you a great deal of debugging time, and they also solve the problem you get when you compile with functions that you use before they are declared. For example, the following code will not compile:

```
#include <stdio.h>
void main()
{
printf("%d\n",add(3));
}
float add(int i, int j)
{
return i+j;
}
```

Why, you might ask, will it compile when add returns an int but not when it returns a float? Because older C compilers default to an int return value. Using a prototype will solve this problem. "Old style" (non-ANSI) compilers allow prototypes, but the parameter list for the prototype must be empty. Old style compilers do no error checking on parameter lists.

C –Recursion:

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

```
void
recursion()function calls itself
}
 int main() {
recursion();
}
```

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.

Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

Number Factorial:

```
#include <stdio.h>
unsigned long long int factorial(unsigned int i) {
```

```
if(i <= 1)
}
    return 1;
}
return i * factorial(i - 1);
}

int main() {
int i = 12;
printf("Factorial of %d is %d\n", i, factorial(i));
return 0;
```

Result
Factorial of 12 is 479001600

Fibonacci Series:

```
#include <stdio.h>

int fibonacci(int i) {

if(i == 0) {
return 0;
}

if(i == 1) {
return 1;
}
return fibonacci(i-1) + fibonacci(i-2);
}

int main() {

int i;
for (i = 0; i < 10; i++) {
printf("%d\t\n", fibonacci(i));
}

return 0;
}
```

When the above code is compiled and executed, it produces the following result −

0
1
1
2
3
5

8
13
21
34

C Standard Library Functions:

C Standard library functions or simply C Library functions are inbuilt functions in C programming.

The prototype and data definitions of these functions are present in their respective header files. To use these functions we need to include the header file in our program.

For example,
If you want to use the printf() function, the header file <stdio.h> should be included.

```
#include <stdio.h>
int main()
{
printf("Catch me if you can.");
}
```

If you try to use printf() without including the stdio.h header file, you will get an error.

Advantages of Using C library functions:

1. They work:

One of the most important reasons you should use library functions is simply because they work. These functions have gone through multiple rigorous testing and are easy to use.

2. The functions are optimized for performance:

Since, the functions are "standard library" functions, a dedicated group of developers constantly make them better. In the process, they are able to create the most efficient code optimized for maximum performance.

3. It saves considerable development time:

Since the general functions like printing to a screen, calculating the square root, and many more are already written. You shouldn't worry about creating them once again.

4. The functions are portable:

With ever-changing real-world needs, your application is expected to work every time, everywhere. And, these library functions help you in that they do the same thing on every computer.

Example: Square root using sqrt() function
Suppose, you want to find the square root of a number.
To can compute the square root of a number, you can use the sqrt() library function. The function is defined in the math.h header file.

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

float num, root;
printf("Enter a number: ");
scanf("%f", &num);

// Computes the square root of num and stores in root.
root = sqrt(num);

printf("Square root of %.2f = %.2f", num, root);
return 0;
}
```

When you run the program, the output will be:

```
Enter a number: 12
Square root of 12.00 = 3.46
```

Library Functions in Different Header Files C Header Files

<assert.h>	Program assertion functions
<ctype.h>	Character type functions
<locale.h>	Localization functions
<math.h>	Mathematics functions
<setjmp.h>	Jump functions
<signal.h>	Signal handling functions
<stdarg.h>	Variable arguments handling functions
<stdio.h>	Standard Input/Output functions
<stdlib.h>	Standard Utility functions
<string.h>	String handling functions
<time.h>	Date time functions

Return Type of a C function:

Every C function must specify the type of data that is being generated. For example, the max function above returns a value of type "double". Inside the function, the line "return X;" must be found, where X is a value or variable containing a value of the given type.

The return statement

When a line of code in a function that says: "return X;" is executed, the function "ends" and no more code in the function is executed. The value of X (or the value in the variable represented by X) becomes the result of the function.

8.2 UNIT END QUESTIONS

1. What is Function ? Explain with Example.

2. What is Function ? What is Function call ?

3. What is the Difference between Call by Value & Call by refrence.

4. What is function Prototype ?

5. What is Recursion ? Explain with Example.

6. Write a C Program to Print Factorial Of User Entered Number.

7. What are the Advantages of C library functions?

*****

Unit III

9

PROGRAM STRUCTURE

Unit Structure

9.1 STORAGE CLASSES

9.1.1 Introduction:

Storage Classes are used to describe the various features of a variable or function. These features include the scope, visibility and life-time which help to trace the existence of a variable during the runtime of a program.

9.1.2 C language uses four storage classes, namely:

1. Auto: This is the default storage class for all variables declared inside a function or a block. Hense, the keyword auto is rarely used while writing programs in C language. Auto variables can be only accessed within the block or function they have been declared and not outside them. These can be accessed within nested blocks within the parent block or function in which the auto variable was declared. They can be accessed outside their scope as well using the concept of pointers given here by pointing to the exact memory location where the variables resides. They are assigned a garbage value by default whenever they are declared.

2. Extern: Extern storage class simply shows that the variable is defined elsewhere and not within the same block where it is used. The value is assigned to it in a different block and this can be overwritten or changed in a different block as well. So an extern variable is a global variable initialized with a legal value where it is declared in order to be used elsewhere. It can be accessed within any function/block. A normal global variable can be made extern as well by placing the _extern' keyword before its declaration or definition in any function or block. This basically signifies that we are not initializing a new variable but instead we are using the global variable only. Then purpose of using extern variables is that they can be accessed between two different files which are part of a large program.

3. Static: This storage class is used to declare static variables which are used while writing programs in C language. Static variables have a property of preserving their value even after they are out of their scope. Hense, static variables preserve their previous value in their previous scope and are not initialized again in the new scope. we can say that they are initialized only once and exist till the end of the program. Hence, no new memory is allocated because they are not re-declared. Their scope is local to the function to which they were defined. Global static variables can be accessed anywhere in the program. By default, 0 value assigned to them by the compiler.

4. Register: This storage class declares register variables which have the same functionality as auto variables. Here, the only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available. This makes the use of register variables to be much faster than that of the variables stored in the memory during the program runtime. If a free register is unavailable, these are then stored in the memory only. Normally few variables which are to be accessed very frequently in a program are declared with the register keyword which improves the running time of the program. An important point to be noted here is that we cannot obtain the address of a register variable using pointers.

To specify the storage class for a variable, the following syntax is to be followed:

Syntax:

```
storage_class var_data_type var_name;

// A C program to demonstrate different storage
// classes
#include <stdio.h>

// declaring the variable which is to be made extern
// an intial value can also be initialized to a

int a;

void autoStorageClass()
{

printf("\nDemonstrating auto class\n\n");

// declaring an auto variable (simply
// writing "int x=32;" works as well)
auto int x = 32;

// printing the auto variable 'x'
```

```c
    printf("Value of the variable 'x'"
    " declared as auto: %d\n",
    x);

    printf("--------------------");
}

void registerStorageClass()
{

    printf("\nDemonstrating register class\n\n");
    // declaring a register variable
    register char c = 'G';

    // declaring a register variable
    register char c = 'G';

    // printing the register variable 'c'
    printf("Value of the variable 'c'"
    " declared as register: %d\n",
    c);

    printf("--------------------");
}

void externStorageClass()
{

    printf("\nDemonstrating extern class\n\n");

    // telling the compiler that the variable
    // z is an extern variable and has been
    // defined elsewhere (above the main
    // function)
    extern int a;

    // printing the extern variables 'a'
    printf("Value of the variable 'a'"
    " declared as extern: %d\n",
    a);

    // value of extern variable a modified
    a = 2;

    // printing the modified values of
    // extern variables 'a'
    printf("Modified value of the variable 'x'"
    " declared as extern: %d\n",
    a );
```

```c
printf("--------------------");
}

void staticStorageClass()
{
int i = 0;

printf("\nDemonstrating static class\n\n");

// using a static variable 'y'
printf("Declaring 'y' as static inside the loop.\n"
"But this declaration will occur only"
" once as 'y' is static.\n"
"If not, then every time the value of 'y' "

"will be the declared value 5"
" as in the case of variable 'p'\n");

printf("\nLoop started:\n");
for (i = 1; i < 5; i++) {

// Declaring the static variable 'y'
static int y = 5;

// Declare a non-static variable 'p'
int p = 10;

// Incrementing the value of y and p by 1
y++;
p++;

// printing value of y at each iteration
printf("\nThe value of 'y', "
"declared as static, in %d "
"iteration is %d\n",
i, y);

// printing value of p at each iteration
printf("The value of non-static variable 'p', "
"in %d iteration is %d\n",
i, p);
}
printf("\nLoop ended:\n");
printf("--------------------");

}

int main()
{
```

```
printf("A program to demonstrate"
" Storage Classes in C\n\n");

// To demonstrate auto Storage Class
autoStorageClass();

// To demonstrate register Storage Class
registerStorageClass();

// To demonstrate extern Storage Class
externStorageClass();

// To demonstrate static Storage Class

staticStorageClass();

// exiting
printf("\n\nStorage Classes demonstrated");

return 0;
}
```

Output:

A program to demonstrate Storage Classes in C

Demonstrating auto class	Value of the variable 'x' declared	'as auto:	32
Demonstrating register class	Value of the variable 'c' declared	'as register:	71
Demonstrating extern class	Value of the variable 'x' declared	as extern:	
Demonstrating extern class	Value of the variable 'x' declared	as extern x'	0
	Modified value of the variable	declared as extern::	2

Demonstrating static class
Declaring 'y' as static inside the loop. But this declaration will occur only once as 'y' is static.If not, then every time the value of 'y' will be the declared value 5 as in the case of variable 'p'
Loop started:
The value of 'y', declared as static, in 1 iteration is 6 The value of non static variable 'p', in 1 iteration is 11
The value of 'y', declared as static, in 1 iteration is 6 The value of non static variable 'p', in 1 iteration is 11
Loop ended:

9.2 MULTI-FILE PROGRAMS

In a program consisting of many different functions, it is convenient to place each function in a separate file, and then use the make utility to compile each file separately and link them together to produce an executable.

There are a some rules associated with multi-file programs. As a given file is initially compiled *separately*, all symbolic constants which appear in that file must be defined at its start. All referenced library functions must be accompanied by the appropriate references to header files. Any referenced user-defined functions must have their prototypes at the start of the file. all global variables used in the file must be declared at its start. This usually means that definitions for common symbolic constants, header files for common library functions, prototypes for common user-defined functions, and declarations for common global variables will appear in *multiple* files. Note that a given global variable can only be initialized in *one* of its declaration statements, which is regarded as the *true declaration* of that variable . Indeed, the other declarations, which we shall term *definitions*, must be preceded by the keyword extern to distinguish them from the true declaration.

As an example, the program printfact.c, break it up into multiple files, each containing a single function. The files in question are called main.c and fact.c. The listings of the two files which make up the program are as follows:

```
/* main.c */
/*
Program to print factorials of all integers
between 0 and 20
*/

#include <stdio.h>

/* Prototype for fucntion factorial() */
void factorial();

/* Global variable declarations */
int j;
double fact;

int main()
{
/* Print factorials of all integers between 0 and 10 */
for (j = 0; j <= 20; ++j)
{
factorial();
```

```c
printf("j = %3d factorial(j) = %12.3e\n", j, fact);
}
return 0;
}
```

and

```c
/* fact.c */
/*
Function to evaluate factorial (in floating point form)
of non-negative integer j. Result stored in variable fact.
*/

#include <stdio.h>
#include <stdlib.h>

/* Global variable definitions */
extern int j;
extern double fact;

void factorial()
{
int count;

/* Abort if j is negative integer */
if (j < 0)
{
printf("\nError: factorial of negative integer not defined\n");
exit(1);
}

/* Calculate factorial */
for (count = j, fact = 1.; count > 0; --count) fact *= (double) count;

return;
}
```

9.3 MORE LIBRARY FUNCTIONS

We can make use of these library functions to get the pre-defined output instead of writing our own code to get those outputs.

- These library functions are created by the persons who designed and created C compilers.

- All C standard library functions are declared in many header files which are saved as file_name.h.

- Actually, function declaration, definition for macros are given in all header files.

- We are including these header files in our C program using ''#include<file_name.h>'' command to make use of the functions those are declared in the header files.

- When we include header files in our C program using ''#include<filename.h>'' command, all C code of the header files are included in C program. Then, this C program is compiled by compiler and executed.

List of most used header files in C programming language:

Check the below table to know all the C library functions and header files in which they are declared.

Header file	Description
stdio.h	This is standard input/output header file in which Input/Output functions are declared
conio.h	This is console input/output header file
string.h	All string related functions are defined in this header file
stdlib.h	This header file contains general functions used in C programs
math.h	All maths related functions are defined in this header file
time.h	This header file contains time and clock related functions
ctype.h	All character handling functions are defined in this header file
stdarg.h	Variable argument functions are declared in this header file
signal.h	Signal handling functions are declared in this file
setjmp.h	This file contains all jump functions
locale.h	This file contains locale functions
errno.h	Error handling functions are given in this file
assert.h	This contains diagnostics functions

9.4 UNIT END QUESTIONS

1. Explain Automatic storage class specifier.
2. Explain static storage class
3. Explain Register storage class
4. Explain extern storage class

*****

<div align="right">**10**</div>

PREPROCESSOR

Unit structure

10.1 PREPROCESSOR

10.1.1 Introduction:

Preprocessor was introduced to C around 1973 at the urging of Alan Snyder and also in recognition of the usefulness of the file-inclusion mechanisms available in BCPL and PL/I. Its original version allowed only to include files and perform simple string replacements: #include and #define of parameterless macros. Soon after that, it was extended, mostly by Mike Lesk and then by John Reiser, to incorporate macros with arguments and conditional compilation.[2]

The C Preprocessor is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation.

10.2 FEATURES

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column.

The C preprocessor provides four separate facilities:

- Inclusion of header files. These are files of declarations that can be substituted into your program.
- Macro expansion. You can define macros, which are abbreviations for arbitrary fragments of C code, and then the C preprocessor will replace the macros with their definitions throughout the program.

- Conditional compilation. Using special preprocessing directives, you can include or exclude parts of the program according to various conditions.
- Line control. If you use a program to combine or rearrange source files into an intermediate file which is then compiled, you can use line control to inform the compiler of where each source line originally came from.

10.3 #DEFINE AND #INCLUDE

10.3.1 #define Directive (macro definition):

Description:

In the C Programming Language, the #define directive allows the definition of macros within your source code. These macro definitions allow constant values to be declared for use throughout your code.

Macro definitions are not variables and cannot be changed by your program code like variables. You generally use this syntax when creating constants that represent numbers, strings or expressions.

Syntax:

The syntax for creating a constant using #define in the C language is:
#define CNAME value
OR
#define CNAME (expression)
CNAME

The name of the constant. Most C programmers define their constant names in uppercase, but it is not a requirement of the C Language.

value

The value of the constant.

expression

Expression whose value is assigned to the constant. The expression must be enclosed in parentheses if it contains operators.

Note:
- Do NOT put a semicolon character at the end of #define statements. This is a common mistake.

Example
Let's look at how to use #define directives with numbers, strings, and expressions.

Number

The following is an example of how you use the #define directive to define a numeric constant:

#define AGE 10

In this example, the constant named AGE would contain the value of 10.

String:

You can use the #define directive to define a string constant.

For example:

#define NAME "TechOnTheNet.com"

In this example, the constant called NAME would contain the value of "TechOnTheNet.com".

Below is an example C program where we define these two constants:
#include <stdio.h>

#define NAME "TechOnTheNet.com"
#define AGE 10
int main()

{
printf("%s is over %d years old.\n", NAME, AGE);
return 0;
}

This C program would print the following:

TechOnTheNet.com

10.3.2 The `#include' Directive:

Both user and system header files are included using the preprocessing directive `#include'. It has three variants:

#include <file>

This variant is used for system header files. It searches for a file named file in a list of directories specified by you, then in a standard list of system directories. You specify directories to search for header files with the command option `-I' (see section 1.9 Invoking the C Preprocessor). The option `-nostdinc' inhibits searching the standard system directories; in this case only the directories you specify are searched.

135

The parsing of this form of `#include' is slightly special because comments are not recognized within the `<...>'. Thus, in `#include <x/*y>' the `/*' does not start a comment and the directive specifies inclusion of a system header file named `x/*y'.

Of course, a header file with such a name is unlikely to exist on Unix, where shell wildcard features would make it hard to manipulate.

The argument file may not contain a `>' character. It may, however, contain a `<' character.

#include "file"

This variant is used for header files of your own program. It searches for a file named file first in the current directory, then in the same directories used for system header files. The current directory is the directory of the current input file. It is tried first because it is presumed to be the location of the files that the current input file refers to. (If the `-I' option is used, the special treatment of the current directory is inhibited.)

The argument file may not contain `"' characters. If backslashes occur within file, they are considered ordinary text characters, not escape characters. None of the character escape sequences appropriate to string constants in C are processed. Thus, `#include "x\n\\y"' specifies a filename containing three backslashes. It is not clear why this behavior is ever useful, but the ANSI standard specifies it.

#include anything else

This variant is called a computed #include. Any `#include' directive whose argument does not fit the above two forms is a computed include. The text anything else is checked for macro calls, which are expanded (see section 1.4 Macros). When this is done, the result must fit one of the above two variants--in particular, the expanded text must in the end be surrounded by either quotes or angle braces.

This feature allows you to define a macro which controls the file name to be used at a later point in the program. One application of this is to allow a site-specific configuration file for your program to specify the names of the system include files to be used. This can help in porting the program to various operating systems in which the necessary system header files are found in different places.

[
[

10.4 DIRECTIVES AND MACROS

The following section lists down all the important preprocessor directives

Sr.No.	Directive & Description
1 #define	Substitutes a preprocessor macro.
2. #include	Inserts a particular header from another file.
3. #undef	Undefines a preprocessor macro.
4. #ifdef	Returns true if this macro is defined.
5. #ifndef	Returns true if this macro is not defined.
6. #if	Tests if a compile time condition is true.
7. #else	The alternative for #if.
8. #elif	#else and #if in one statement.
9. #endif	Ends preprocessor conditional.
10. #error	Prints error message on stderr.
11. #pragma	Issues special commands to the compiler, using a standardized method.

Preprocessors Examples:

To understand various directives refer this examples

#define MAX_ARRAY_LENGTH 20

This directive tells to replace instances of MAX_ARRAY_LENGTH with 20. Use #define for constants to increase readability.

#include <stdio.h>
#include "myheader.h"

These directives tell to get stdio.h from System Libraries and add the text to the current source file. The next line tells to get myheader.h from the local directory and add the content to the current source file.

#undef FILE_SIZE
#define FILE_SIZE 45

It tells to undefine existing FILE_SIZE and define it as 45.

#ifndef MESSAGE
#define MESSAGE "You wish!"
#endif

It tells to define MESSAGE only if MESSAGE isn't already defined.
#ifdef DEBUG
/* Your debugging statements here */

#endif

It tells to process the statements enclosed if DEBUG is defined. This is useful if you pass the -DDEBUG flag to the gcc compiler at the time of compilation. This will define DEBUG, so you can turn debugging on and off on the fly during compilation.

Predefined Macros:

ANSI C defines a number of macros. Although each one is available for use in programming, the predefined macros should not be directly modified.

Sr.No.
Macro & Description
1 __DATE__
The current date as a character literal in "MMM DD YYYY" format.

2 __TIME__
The current time as a character literal in "HH:MM:SS" format.

3 __FILE__
This contains the current filename as a string literal.

4 __LINE__
This contains the current line number as a decimal constant.

5 __STDC__
Defined as 1 when the compiler complies with the ANSI standard.

Let's try the following example −

Live Demo:

#include <stdio.h>

int main() {

printf("File :%s\n", __FILE__);
printf("Date :%s\n", __DATE__);
printf("Time :%s\n", __TIME__);
printf("Line :%d\n", __LINE__);
printf("ANSI :%d\n", __STDC__);

}

When the above code in a file test.c is compiled and executed, it produces the following result −

File :test.c
Date :July 5 2018
Time :03:45:25
Line :8
ANSI :1

Preprocessor Operators:

The C preprocessor offers the following operators to help create macros −

The Macro Continuation (\) Operator:

A macro is normally confined to a single line. The macro continuation operator (\) is used to continue a macro that is too long for a single line.

For example:

#define message_for(a, b) \
printf(#a " and " #b ": We love you!\n")

The Stringize (#) Operator:

The stringize or number-sign operator ('#'), when used within a macro definition, converts a macro parameter into a string constant. This operator may be used only in a macro having a specified argument or parameter list. For example −

Live Demo
#include <stdio.h>
#define message_for(a, b) \

printf(#a " and " #b ": We love you!\n")
int main(void) {

message_for(C, D);
return 0;
}

When the above code is compiled and executed, it produces the following result −

C and D: We love you!

The Token Pasting (##) Operator:

The token-pasting operator (##) within a macro definition combines two arguments. It permits two separate tokens in the macro definition to be joined into a single token. For example −

Live Demo:

#include <stdio.h>

```
#define tokenpaster(n) printf ("token" #n " = %d", token##n)
int main(void) {
int token35 = 50;
tokenpaster(35);
return 0;
}
```

When the above code is compiled and executed, it produces the following result

Token35 = 50

It happened so because this example results in the following actual output from the preprocessor −

printf ("token35 = %d", token35);

This example shows the concatenation of token##n into token34 and here we have used both stringize and token-pasting.

The Defined() Operator:

The preprocessor defined operator is used in constant expressions to determine if an identifier is defined using #define. If the specified identifier is defined, the value is true (non-zero). If the symbol is not defined, the value is false (zero). The defined operator is specified as follows –

Live Demo|:

#include <stdio.h>

```
#if !defined (MESSAGE)
#define MESSAGE "You are amazing!"

#endif
int main(void) {
printf("Here is the message: %s\n", MESSAGE);
return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Here is the message: You are amazing!
```

Parameterized Macros:

One of the powerful functions is the ability to simulate functions using parameterized macros. For example, we might have some code to square a number as follows

```
int square(int y) {
return y * y;
}
```

We can rewrite above the code using a macro as follows −

```
#define square(y) ((y) * (y))
```

Macros with arguments must be defined using the #define directive before they can use. The argument list is enclosed in parentheses and must immediately follow the macro name. Spaces are not allowed between the macro name and open parenthesis. For example −

Demo:

```
#include <stdio.h>

#define MAX(x,y) ((x) > (y) ? (x) : (y))

int main(void) {
printf("Max between 30 and 20 is %d\n", MAX(20, 30));
return 0;
}
```

When the above code is compiled and executed, it produces the following result –

Max between 30 and 20 is 2

10.5 UNIT END QUESTIONS

1. What is macro? Summarize the similarities and diffrences between macros and functions

2. What is preprocessor in C Language? Explain #if#else#endif preprocessor directive with suitable example.

3. Write a small program to show the use of macro. rite a small program to show the use of macro.

4. List various preprocessor and explain any two of them

*****

<div align="right">

11

</div>

ARRAY

Unit Structure

11.1 ARRAYS

11.1.1 Introduction:

Arrays a type of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as num0, num1, ..., and num99, you declare one array variable such as numbers and use num[0], num[1], and ..., num[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.
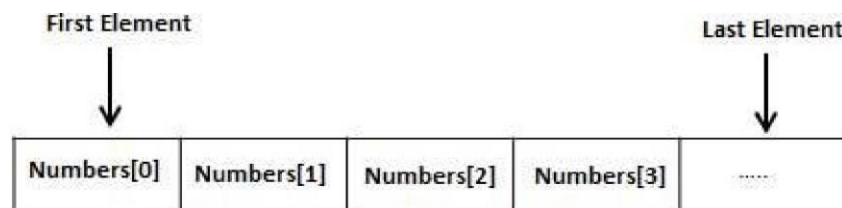
11.2 DECLARING ARRAYS

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows −

type aName [aSize];

This is called a single-dimensional array. The arraySize must be an integer constant greater than zero and type can be any valid C data type. For example, to declare a 10-element array called balance of type double, use this statement −

double balance[20];

Here balance is a variable array which is sufficient to hold up to 10 double numbers.

Initializing Arrays:

You can initialize an array in C either one by one or using a single statement as follows −

double balance[4] = {1000.0, 2.0, 3.4, 7.0};

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

double balance[] = {1000.0, 2.0, 3.4, 7.0};

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array −

balance[3] = 7.0;

The above statement assigns the 4th element in the array with a value of 7.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the array we discussed above –

	0	1	2	3
balance	1000.0	2.0	3.4	7.0

Accessing Array Elements:

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example −

double salary = balance[9];

The above statement will take the 10th element from the array and assign the value to salary variable. The following example Shows how to use all the three above mentioned concepts viz. declaration, assignment, and accessing arrays −

Demo

```
#include <stdio.h>

int main () {

int n[ 10 ]; /* n is an array of 10 integers */
int i,j;

/* initialize elements of array n to 0 */

for ( i = 0; i < 10; i++ ) {
n[ i ] = i + 100; /* set element at location i to i + 100 */
}

/* output each array element's value */
for (j = 0; j < 10; j++ ) {
printf("number[%d] = %d\n", j, n[j] );
}

return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
t[0] = 100
t[1] = 101
t[2] = 102
t[3] = 103
t[4] = 104
t[5] = 105
```

t[6] = 106
t[7] = 107
t[8] = 108
t[9] = 109

11.3 HOW TO PASS ARRAY TO A FUNCTION IN C

Whenever we need to pass a list of elements as argument to any function in C language, it is prefered to do so using an array. But how can we pass an array as argument to a function?

Declaring Function with array as a parameter
There are two possible ways to do so, one by using call by value and other by using call by reference.
1. We can either have an array as a parameter.

int add (int a[]);
2. Or, we can have a pointer in the parameter list, to hold the base address of our array.

int add (int* ptr);

Returning an Array from a function:
We don't return an array from functions, rather we return a pointer holding the base address of the array to be returned. But we must, make sure that the array exists after the function ends i.e. the array is not local to the function.
int* add (int x[])
{
// statements
return x ;
}

Passing arrays as parameter to function:

We will pass a single array element as argument to a function, a one dimensional array to a function and a multidimensional array to a function.

Passing a single array element to a function
We will declare and define an array of integers in main() function and pass one of the array element to a function, which will just print the value of the element.

#include<stdio.h>
void MyArray(int a);

int main()
{

147

```
int mArray[] = { 2, 3, 4 };
MyArray(mArray[2]); //Passing array element myArray[2] only.
return 0;
}
void MyArray(int a)
{
printf("%d", a);
}
Output
4
```

Passing a complete One-dimensional array to a function

let's write a function to find out average of all the elements of the array and print it.

We will only send in the name of the array as argument, which is nothing but the address of the starting element of the array, or we can say the starting memory address.

```
#include<stdio.h>

float findAverage(int marks[]);

int main()
{
float avg;
int marks[] = {99, 90, 96, 93, 95};
avg = findAverage(marks); // name of the array is passed as
        argument.
printf("Average marks = %.1f", avg);
return 0;
}

float findAverage(int marks[])
{
int i, sum = 0;
float avg;
for (i = 0; i <= 4; i++) {
sum += marks[i];
}
avg = (sum / 5);
return avg;
}
```

11.4 MULTIDIMENSIONAL ARRAYS

- Multi-dimensional arrays are declared by providing more than one set of square [ ] brackets after the variable name in the declaration statement.

- One dimensional arrays do not require the dimension to be given if the array is to be completely initialized. By analogy, multi-dimensional arrays do not require **the first** dimension to be given if the array is to be completely initialized. All dimensions after the first must be given in any case.

- For two dimensional arrays, the first dimension is commonly considered to be the number of rows, and the second dimension the number of columns. We will use this convention when discussing two dimensional arrays.

- Two dimensional arrays are considered by C/C++ to be an array of (single dimensional arrays). For example, "int numbers[5][6]" would refer to a single dimensional array of 5 elements, wherein each element is a single dimensional array of 6 integers. By extension, "int numbers[12][5][6]" would refer to an array of twelve elements, each of which is a two dimensional array, and so on.

- Another way of looking at this is that C stores two dimensional arrays by rows, with all elements of a row being stored together as a single unit. Knowing this can sometimes lead to more efficient programs.

- Multidimensional arrays may be completely initialized by listing all data elements within a single pair of curly {} braces, as with single dimensional arrays.

- It is better programming practice to enclose each row within a separate subset of curly {} braces, to make the program more readable. This is required if any row other than the last is to be partially initialized. When subsets of braces are used, the last item within braces is not followed by a comma, but the subsets are themselves separated by commas.

- Multidimensional arrays may be partially initialized by not providing complete initialization data. Individual rows of a multidimensional array may be partially initialized, provided that subset braces are used.

- Individual data items in a multidimensional array are accessed by fully qualifying an array element. Alternatively, a smaller dimensional array may be accessed by partially qualifying the array name. For example, if "data" has been declared as a three dimensional array of floats, then data[1][2][5] would refer to a float, data[1][2] would refer to a one-dimensional array of floats, and data[1] would refer to a two-dimensional array of floats. The reasons for this and the incentive to do this relate to memory-management issues that are beyond the scope of these notes.

Sample Program Using 2-D Arrays
/* Sample program Using 2-D Arrays */

#include <stdlib.h>
#include <stdio.h>

```c
int main( void ) {

/* Program to add two multidimensional arrays */
/* Written May 1995 by George P. Burdell */

int a[ 2 ][ 3 ] = { { 5, 6, 7 }, { 10, 20, 30 } };
int b[ 2 ][ 3 ] = { { 1, 2, 3 }, { 3, 2, 1 } };

int sum[ 2 ][ 3 ], row, column;

/* First the addition */
for( row = 0; row < 2; row++ )
for( column = 0; column < 3; column++ )
sum[ row ][ column ] =
a[ row ][ column ] + b[ row ][ column ];

/* Then print the results */

printf( "The sum is: \n\n" );

for( row = 0; row < 2; row++ ) {
for( column = 0; column < 3; column++ )
printf( "\t%d", sum[ row ][ column ] );
printf( '\n' ); /* at end of each row */
}

return 0;

}
```

Passing a Multi-dimensional array to a function

We will pass the name of the array as argument.

```c
#include<stdio.h>

void displayArray(int arr[3][3]);

int main()
{
int arr[3][3], i, j;
printf("Please enter 9 numbers for the array: \n");
for (i = 0; i < 3; ++i)
{

for (j = 0; j < 3; ++j)
{

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

```
}
}
// passing the array as argument
displayArray(arr);
return 0;
}
void displayArray(int arr[3][3])
{

int i, j;
printf("The complete array is: \n");
for (i = 0; i < 3; ++i)
{
// getting cursor to new line
printf("\n");
for (j = 0; j < 3; ++j)
{
// \t is used to provide tab space
printf("%d\t", arr[i][j]);
}
}
}
```

Please enter 9 numbers for the array: 1 2 3 4 5 6 7 8 9 The complete array is: 1 2 3 4 5 6 7 8 9

11.5 ARRAYS AND STRINGS

In C programming, a string is a sequence of characters terminated with a null character \0. For example:

char c[] = "c string";

When the compiler encounters a sequence of characters enclosed in the double quotation marks, it appends a null character \0 at the end by default.

c		s	t	r	i	n	g	\0

How to declare a string?
• char s[5];
• Strings can also be declared using pointer.
char *p;

s[0]	s[1]	s[2]	s[3]	s[4]

How to initialize strings?

We can initialize strings in a number of ways.
• char c[] = "abcd"; or

• char c[5] = "abcd"; or

• char c[] = {'a', 'b', 'c', 'd', '\0'}; or

• char c[5] = {'a', 'b', 'c', 'd', '\0'};

c[0]	c[1]	c[2]	c[3]	c[4]
a	b	c	d	\0

Let's take another example:
• char c[5] = "abcde";

• Here, we are trying to assign 6 characters (the last character is '\0') to a char array having 5 characters. This is bad and you should never do this.

Read String from the user
• We can use the scanf() function to read a string.
• The scanf() function reads the sequence of characters until it encounters whitespace (space, newline, tab etc.).

Example 1: scanf() to read a string
```
#include <stdio.h>

int main()

{

char name[20];

printf("Enter name: ");

scanf("%s", name);

printf("Your name is %s.", name);

return 0;
}
```

• **Output**
Enter name: Anita Jaykar

152

Your name is Anita.

Even though Anita Jaykar was entered in the above program, only ─Anita"
was stored in the name string. It's because there was a space after Anita.

How to read a line of text?
You can use the gets() function to read a line of string. And, you can use
puts() to display the string.

Example 2: gets() and puts()
```c
#include <stdio.h>
int main()

{

char name[30]
;
printf("Enter name: ");

gets(name); // read string

printf("Name: ");

puts(name); // display string

return 0;

}
```

Output:

Enter name: Anita Jain

Name: Anita Jain

Passing Strings to Functions
• Strings can be passed to a function in a similar way as arrays.

Example 3: Passing string to a Function:

```c
#include <stdio.h>

void displayString(char str[]);

int main()

{

char str[50];
```

153

```
printf("Enter string: ");

gets(str);

displayString(str); // Passing string to a function.

return 0;

}
void displayString(char str[])
{

printf("String Output: ");

puts(str);

}
```

Commonly Used String Functions:
• **strlen() - calculates the length of a string**

• **strcpy() - copies a string to another**

• **strcmp() - compares two strings**

• **strcat() - concatenates two strings**

String Manipulation:

string.h
•   C supports a large number of string handling functions in the standard library "string.h".

•   **Note:** Though, gets() and puts() function handle strings, both these functions are defined in "stdio.h" header file.

Few commonly used string handling functions

Function	Work of Function
Strlen ()	Calculates the length of string
Strcpy()	Copies the string to another string
Strcat(0	Concatanates (joins) two strings
Strcmp()	Compares two string
Strlwr()	Converts string to lower case
Strupr ()	Converts string to upper case

Strlen ():
•   In C, strlen() function calculates the length of string. It takes only one argument, i.e, string name.

- Defined in Header File <string.h>
- Syntax of strlen() :

temp_variable = strlen(string_name);

Function strlen() returns the value of type integer.

Program to Find the Length of a String:

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

{
char a[20]="Program";

char b[20]={'P','r','o','g','r','a','m','\0'};

char c[20];

printf("Enter string: ");

gets(c);

printf("Length of string a=%d \n",strlen(a));

//calculates the length of string before null charcter.

printf("Length of string b=%d \n",strlen(b));

printf("Length of string c=%d \n",strlen(c));

return 0;

}
```

Enter string: String

Length of string a=7

Length of string b=7

Length of string c=6

Strcpy()

- Function strcpy() copies the content of one string to the content of another string.
- It takes two arguments.
- Defined in Header File <string.h>
- Syntax of strcpy() :

strcpy(destination,source);

- Here, source and destination are both the name of the string. This statement, copies the content of string source to the content of string destination.

Example of strcpy():
```
#include <stdio.h>
#include <string.h>

int main()

{

char a[10],b[10];

printf("Enter string: ");

gets(a);

strcpy(b,a); //Content of string a is copied to string b. printf("Copied string: ");

puts(b);

return 0;
}
```

• Enter string: Anita

• Copied string: Anita

strcat():
- In C programming, strcat() concatenates(joins) two strings.
- It takes two arguments, i.e, two strings and resultant string is stored in the first string specified in the argument.

Syntax of strcat()
```
• strcat(first_string,second_string);
#include <stdio.h>

#include <string.h>

int main()

{

char str1[]="my name is ", str2[]=―Anita";
```

strcat(str1,str2); //concatenates str1 and str2 and resultant string is stored in str1.

puts(str1);
puts(str2);

return 0;

}

Output :
My name is Anita.
Anita

Strcmp():
- In C programming, strcmp() compares two string and returns value 0, if the two strings are equal.
- Function strcmp() takes two arguments, i.e, name of two string to compare.
- strcmp(string1,string2);
- #include <stdio.h>
- #include <string.h>
- int main()
- {
- char str1[30],str2[30];
- printf("Enter first string: ");
- gets(str1);
- printf("Enter second string: ");
- gets(str2);
- if(strcmp(str1,str2)==0)
- printf("Both strings are equal");
- else printf("Strings are unequal");
- return 0;
- }

Output:
Enter first string: Apple
Enter second string: Apple
Both strings are equal.

11.6 UNIT END QUESTIONS

1. List the characteristics of Arrays
2. What are the main elements of array declaration.
3. Explain two dimensional array with example.
4. What is string? Explain about a)strcmp() b)strlen() function.

5. Write a program that performs multiplication of two matrices
6. What is multidimensional array
7. Write a program to find the largest value that is stored in the array
8. Write a program that performs addition and subtraction of matrices.
9. Write a program to arrange n numbers stored in the array in ascending order.-

*****

UNIT V

12

POINTERS

Unit Structure
12.0   Objectives
12.1   Fundamentals Of Pointers
12.2   Address Operations
12.4   Pointer Assignment
12.5   Pointer Arithmetic

12.0 OBJECTIVES

- To understand the meaning and need of pointers
- To learn the syntax to declare, initialize and use pointers
- To understand the addressing operations using & and *
- To learn pointer arithmetic

12.1 FUNDAMENTALS OF POINTERS

Pointers are variables that hold a memory location. One can access the value of the variable pointed to using the dereferencing operator *. A pointer is a value that designates the address (i.e., the location in memory), of some value.
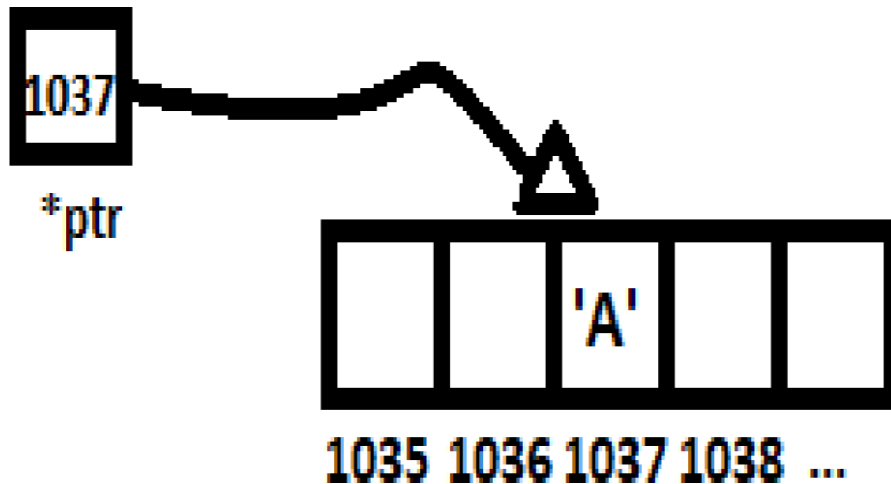
Fig 1. Pointer *ptr stores the address of a character value 'A'

Pointers can reference any data type, even functions.

Advantages of pointers:
1) Pointer reduces the code and improves the performance, it is used to retrieving strings, trees, etc. and used with arrays, structures, and functions.
2) We can return multiple values from a function using the pointer.
3) It makes you able to access any memory location in the computer's memory.

12.2 ADDRESS OPERATIONS

Every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which prints the address of the variables defined

```
#include <stdio.h>

int main () {
int va1;
char var2[10];

printf("Address of var1 variable: %d\n", &var1 );
printf("Address of var2 variable: %d\n", &var2 );
return 0;
}
```

Output:

```
Address of var1 variable: 3400
Address of var2 variable: 9656
```

Address operators:

There are two important operators which are highly required, if you are working with the pointers. Without these operators, we cannot work with the pointers.

The operators are:
- The * Operator (Dereference Operator or Value at Operator)
- The & Operator (Address Of Operator)

1) The * Operator (Dereference Operator or Value at Operator)

"Dereference Operator" or "Value at" Operator denoted by asterisk character (*), * is a unary operator which performs two operations with the pointer (which is used for two purposes with the pointers).

- To declare a pointer
- To access the stored value of the memory (location) pointed by the pointer

2) The & Operator (Address of Operator):

The "Address Of" Operator denoted by the ampersand character (&), & is a unary operator, which returns the address of a variable.

After declaration of a pointer variable, we need to initialize the pointer with the valid memory address; to get the memory address of a variable Address Of" (&) Operator is used.

12.3 POINTER TYPE DECLARATION

```
int *p1; /*Pointer to an integer variable*/
double *p2; /*Pointer to a variable of data type double*/
char *p3; /*Pointer to a character variable*/
float *p4; /*pointer to a float variable*/
```

In the above snippet we have declared p1 as an pointer which would point to an integer variable. Correspondingly p2 , p3 and p4 would point to double, char and float variables respectively.

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

12.4 POINTER ASSIGNMENT

Every pointer stores the address of a variable which it points to. The value is a number that represents the memory address of the variable which it points to.

Pointers (that is, pointer values) are generated with the ``address-of'' operator &, which we can also think of as the ``pointer-to'' operator. We demonstrate this by declaring (and initializing) an int variable i, and then setting ip to point to it:

```
int  *ptr;
int i = 5;
ptr = &i;
```

The assignment expression ip = &i; contains both parts of the ``two-step process'': &i generates a pointer to i, and the assignment operator assigns the new pointer to (that is, places it ``in'') the variable ip. Now ip ``points to'' i, which we can illustrate with this picture:

i is a variable of type int, so the value in its box is a number, 5. ip is a variable of type pointer-to-int, so the ``value'' in its box is an arrow pointing at another box. Referring once again back to the ``two-step process'' for setting a pointer variable: the & operator draws us the arrowhead pointing at i's box, and the assignment operator =, with the pointer variable ip on its left, anchors the other end of the arrow in ip's box.

Another Example:
```
#include<stdio.h>
int main(){
int number=50;
int *p;
p=&number;//stores the address of number variable
printf("Address of p variable is %x \n",p);
printf("Value of p variable is %d \n",*p);
```

```
return 0;
}
```

Output:

```
Address of number variable is fff4
Address of p variable is fff4
Value of p variable is 50
```

Explanation:

p contains the address of the number therefore printing p gives the address of number. As we know that * is used to dereference a pointer therefore if we print *p, we will get the value stored at the address contained by p.

12.5 POINTER ARITHMETIC

C allows you to perform some arithmetic operations on pointers. (Not every operation is allowed.)

Unary Pointer Arithmetic Operators
Operator ++: Adds sizeof(datatype) number of bytes to pointer, so that it points to the next entry of the datatype.

Operator −−: Subtracts sizeof(datatype) number of bytes to pointer, so that it points to the next entry of the datatype.

```
#include <stdio.h>
int main()
{
int *ptrn;
long *ptrlng;
ptrn++; //increments by sizeof(int) (4 bytes)
ptrlng++; //increments by sizeof(long) (8 bytes)
return 0;
}
```

Similarly, use of -- operator decrements the value of the pointer by 'n' bytes, where n is the size in bytes of the variable datatype.

We can also use the binary + and – operators. It is important to note that we cannot add to pointers to each other because that would mean adding 2 addresses. However, if we use the following:

```
ptr2 = ptr1 + 8;
```

162

That would mean ptr2 would point to the memory location 8 bytes ahead of ptr1.

Incrementing a Pointer:

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array

```
#include <stdio.h>
const int MAX = 3;

int main () {

int var[] = {10, 100, 200};
int i, *ptr;
/* let us have array address in pointer */
ptr = var;

for ( i = 0; i < MAX; i++) {

printf("Address of var[%d] = %x\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );

/* move to the next location */
ptr++;
}
return 0;
}
```

Output:

```
Address of var[0] = bf882b30
Value of var[0] = 10
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

Decrementing a Pointer:

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below

```
#include <stdio.h>

const int MAX = 3;

int main () {
```

```
int var[] = {10, 100, 200};
int i, *ptr;

/* let us have array address in pointer */
ptr = &var[MAX-1];

for ( i = MAX; i > 0; i--) {

printf("Address of var[%d] = %x\n", i-1, ptr );
printf("Value of var[%d] = %d\n", i-1, *ptr );

/* move to the previous location */
ptr--;
}

return 0;
}
```

Output:

```
Address of var[2] = bfedbcd8
Value of var[2] = 200
Address of var[1] = bfedbcd4
Value of var[1] = 100
Address of var[0] = bfedbcd0
Value of var[0] = 10
```

Pointer Comparisons:

Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

The following program modifies the previous example − one by incrementing the variable pointer so long as the address to which it points is either less than or equal to the address of the last element of the array, which is &var[MAX - 1]

```
#include <stdio.h>

const int MAX = 3;

int main () {

int var[] = {10, 100, 200};
int i, *ptr;
```

164

```
/* let us have address of the first element in pointer */
ptr = var;
i = 0;

while ( ptr <= &var[MAX - 1] ) {

printf("Address of var[%d] = %x\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );

/* point to the next location */
ptr++;
i++;

}

return 0;

}
```

Output:

```
Address of var[0] = bfdbcb20
Value of var[0] = 10
Address of var[1] = bfdbcb24
Value of var[1] = 100
Address of var[2] = bfdbcb28
Value of var[2] = 200
```

12.6 UNIT END QUESTIONS

1. What if a pointer and the advantage of declaring void pointers?
2. Difference between pass by reference and pass by value?
3. When should we use pointers in a C program?
4. What happens when we use ++ and -- operator on an integer pointer?

*****

13

ADVANCED POINTERS

Unit Structure

13.0 OBJECTIVES

- To build on the fundamentals of pointers learnt in the previous chapter
- To understand the use of pointers as parameters to functions
- To use of pointers with reference to an array
- To study the use of pointer arrays
- To learn how to pass function pointers as parameters

13.1 FUNCTIONS & POINTERS

C programming allows passing a pointer to a function. To do so, simply declare the function parameter as a pointer type. Since pointers are also variables, they can be passed

- As input parameters to functions
- As return values from functions

The call by reference method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.

Following is a simple example where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function

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

void getSeconds(unsigned long *par);

int main () {
```

166

```
unsigned long sec;
getSeconds( &sec );

/* print the actual value */
printf("Number of seconds: %ld\n", sec );

return 0;
}

void getSeconds(unsigned long *par) {
/* get the current number of seconds */
*par = time( NULL );
return;
}
```

Output:
Number of seconds :1394450468

Pass by Value vs Pass by Reference:

To pass a value by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function swap(), which exchanges the values of the two integer variables pointed to, by their arguments.

```
/* function definition to swap the values */
void swap(int *x, int *y) {

int temp;
temp = *x; /* save the value at address x */
*x = *y; /* put y into x */
*y = temp; /* put temp into y */

return;
}
```

Let Let us now call the function swap() by passing values by reference as in the following example:

Output:

Before swap, value of a :	100
Before swap, value of b :	200
After swap, value of a :	200
After swap, value of b :	100

167

It shows that the change has reflected outside the function as well, unlike call by value where the changes do not reflect outside the function.

13.2 ARRAYS & POINTERS

An array is a block of sequential data. Let's write a program to print addresses of array elements.

```
#include <stdio.h>
int main() {
int x[4];
int i;

for(i = 0; i < 4; ++i) {
printf("&x[%d] = %p\n", i, &x[i]);
}

printf("Address of array x: %p", x);

return 0;

}
```

Output:

&x[0] = 1450734448
&x[1] = 1450734452
&x[2] = 1450734456
&x[3] = 1450734460
Address of array x: 1450734448

There is a difference of 4 bytes between two consecutive elements of array x. It is because the size of int is 4 bytes (on our compiler).

Notice that, the address of &x[0] and x is the same. It's because the variable name x points to the first element of the array.

Relation between arrays and pointers:

From the above example, it is clear that &x[0] is equivalent to x. And, x[0] is equivalent to *x.

Similarly,

&x[1] is equivalent to x+1 and x[1] is equivalent to *(x+1).

168

&x[2] is equivalent to x+2 and x[2] is equivalent to *(x+2).
...

Basically, &x[i] is equivalent to x+i and x[i] is equivalent to *(x+i).

Example 1: Pointers and Arrays

```
#include <stdio.h>
int main() {
int i, x[6], sum = 0;
printf("Enter 6 numbers: ");
for(i = 0; i < 6; ++i) {
// Equivalent to scanf("%d", &x[i]);
scanf("%d", x+i);
// Equivalent to sum += x[i]
sum += *(x+i);
}
printf("Sum = %d", sum);
return 0;
}
```

Output:
Enter 6 numbers:
2
3
4
4
13
4
Sum = 29

Here, we have declared an array x of 6 elements. To access elements of the array, we have used pointers.

In most contexts, array names decay to pointers. In simple words, array names are converted to pointers. That's the reason why you can use pointers to access elements of arrays. However, you should remember that pointers and arrays are not the same.

Example 2: Arrays and Pointers

```
#include <stdio.h>
int main() {
int x[5] = {1, 2, 3, 4, 5};
int* ptr;

// ptr is assigned the address of the third element
ptr = &x[2];
```

```
printf("*ptr = %d \n", *ptr);                    // 3
printf("*(ptr+1) = %d \n", *(ptr+1));            // 4
printf("*(ptr-1) = %d", *(ptr-1));               // 2

return 0;
}
```

Output:

```
*ptr = 3
*(ptr+1) = 4
*(ptr-1) = 2
```

In this example, &x[2], the address of the third element, is assigned to the ptr pointer. Hence, 3 was displayed when we printed *ptr.

And, printing *(ptr+1) gives us the fourth element. Similarly, printing *(ptr-1) gives us the second element.

13.3 POINTER ARRAYS

Before we understand the concept of arrays of pointers, let us consider the following example, which uses an array of 3 integers

```
#include <stdio.h>

const int MAX = 3;

int main () {

int var[] = {10, 100, 200};
int i;

for (i = 0; i < MAX; i++) {
printf("Value of var[%d] = %d\n", i, var[i] );
}

return 0;
}
```

Output:

```
Value of var [0] = 10
Value of var [1] = 100
Value of var [2] = 200
```

There may be a situation when we want to maintain an array, which can store pointers to an int or char or any other data type available. Following is the declaration of an array of pointers to an integer

int *ptr[MAX];

It declares ptr as an array of MAX integer pointers. Thus, each element in ptr, holds a pointer to an int value. The following example uses three integers, which are stored in an array of pointers, as follows

```
#include <stdio.h>

const int MAX = 3;

int main () {

int var[] = {10, 100, 200};
int i, *ptr[MAX];

for ( i = 0; i < MAX; i++) {
ptr[i] = &var[i]; /* assign the address of integer. */
}

for ( i = 0; i < MAX; i++) {
printf("Value of var[%d] = %d\n", i, *ptr[i] );
}

return 0;

}
```

Output:

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

You can also use an array of pointers to character to store a list of strings as follows

```
#include <stdio.h>

const int MAX = 4;

int main () {

char *names[] = {

"RAMESH",
"REENA",
"SHYAM",
"ALI"
};
```

```
int i = 0;

for ( i = 0; i < MAX; i++) {
printf("Value of names[%d] = %s\n", i, names[i] );
}
return 0;
}
```

Output:

Value of names[0] = RAMESH
Value of names[1] = REENA
Value of names[2] = SHYAM
Value of names[3] = ALI

13.4 PASSING FUNCTIONS TO OTHER FUNCTIONS

A simple prototype for a function which takes a function parameter (sometimes called a formal parameter), is something like this:

void myfunction(void (*f)(int));

This states that a parameter f will be a pointer (*f) to the function myFunction, which has a void return type and which takes just a single int parameter.

In lay man's terms, my Function takes an argument of a function type void, that returns a type void, and takes an int as an argument;

(void (*f)(int)).

A simple example:

```
#include <stdio.h>

void print()
{
printf("Hello World!");
}

void helloworld(void (*f)())
{
f();
}

int main(void)
{
```

```
helloworld(print);
return (0);
}
```

Here, we see that the function named "print " is being passed as a parameter to the function helloworld in main.

Function Pointers:

To understand this better we need to understand function pointers. In C programming language, we can have a concept of Pointer to a function known as function pointer in C. In this tutorial, we will learn how to declare a function pointer and how to call a function using this pointer. To understand this concept, you should have the basic knowledge of Functions and Pointers in C.

Function pointer declaration:

function_return_type(*Pointer_name)(function argument list)

For example:

double (*p2f)(double, char)

Here double is a return type of function, p2f is name of the function pointer and (double, char) is an argument list of this function. Which means the first argument of this function is of double type and the second argument is char type.

Lets understand this with the help of an example: Here we have a function sum that calculates the sum of two numbers and returns the sum. We have created a pointer f2p that points to this function, we are invoking the function using this function pointer f2p.

```
int sum (int num1, int num2)
{
return num1+num2;
}
int main()
{

/* The following two lines can also be written in a single
* statement like this: void (*fun_ptr)(int) = &fun;
*/
int (*f2p) (int, int);
f2p = sum;
//Calling function using function pointer
int op1 = f2p(10, 13);

//Calling function in normal way using function name
int op2 = sum(10, 13);
```

```
printf("Output1: Call using function pointer: %d",op1);
printf("\nOutput2: Call using function name: %d", op2);
return 0;
}
```

Output
Output1: Call using function pointer: 23
Output2: Call using function name: 23

Some points regarding function pointer:

1. As mentioned in the comments, you can declare a function pointer and assign a function to it in a single statement like this:

    void void (*fun_ptr)(int) = &fun;

2. You can even remove the ampersand from this statement because a function name alone represents the function address. This means the above statement can also be written like this:

    void (*fun_ptr)(int) = fun;

13.5 QUESTIONS

1. Difference between pass by reference and pass by value?

2. What is the difference between array of pointers and pointer arrays? give examples.

3. Write a program to print an array of 10 numbers using a pointer to that array. (Do not use array indexes)

4. Write a program to store an array of 10 strings and display them using pointers only.

5. What are function pointers? when are they used?

*****

14

STRUCTURES AND UNIONS

Unit Structure

14.0 OBJECTIVES

- To understand on the concept and need of structures in C
- To understand and implement user defined structures in a C program
- To understand the use of structures with arrays and functions
- To learn pointers to structures
- To learn the usage of unions in C

14.1 INTRODUCTION

When there is a need to store data elements of different types together, arrays are no longer useful. In C, the concept of structures allow developers to group elements of different types together in a structured manner.

Consider a case where a developer needs to store employee details such as name, id, age, address, and salary. The developer could declare 5 separate variables with different data types for each. This would be tedious is we have more than 1 employee because then the developer would have to create 5 variables for each employee and could get too confusing too fast. So, the developer could create a C structure using struct Keyword and assign the name as an employee.

14.2 INITIALIZATION

- Let us now look at how to declare and initialize structures in C. The syntax for declaring a structure is as follows

175

```
struct structureName
{

dataType member1;
dataType member2;
...

};
```

Example:

```
struct Employee

{

int id;

char name[50];

int age;

char address[100];

float salary;
};
```

Note: Members inside the structures will not store any memory location until they are associated with structure variables.

So, we have to create the structure variable before using it. We can declare the C structure variables in multiple ways

Method #1
Create a struct variable after the declaration of structure.

Example:

```
struct Employee

{
int id;
char name[50];
int age;
char address[100];
float salary;
} emp1 , emp2;
```

14.2 ASSIGNMENT

Once we declare the struct variables we need to assign values and use them. This is done using the member access operator . (dot).

Example:
emp1.id   = 191;
emp1.age   = 35;
emp1.salary = 25000;

Lets now look at a complete example which brings the different pieces together. We will consider a structure called distance which store distance in feet and inches. The program will demonstrate how to create a struct called distance and add two variable of the structure type distance.

SAMPLE PROGRAM:

```
// Program to add two distances (feet-inch)
#include <stdio.h>

struct Distance

{
int feet;
float inch;
} dist1, dist2, sum;

int main()
{
printf("1st distance\n");
printf("Enter feet: ");
scanf("%d", &dist1.feet);

printf("Enter inch: ");
scanf("%f", &dist1.inch);
printf("2nd distance\n");

printf("Enter feet: ");
scanf("%d", &dist2.feet);

printf("Enter inch: ");
scanf("%f", &dist2.inch);

// adding feet
sum.feet = dist1.feet + dist2.feet;
// adding inches
sum.inch = dist1.inch + dist2.inch;
// changing to feet if inch is greater than 12
```

```
while (sum.inch >= 12)
{
++sum.feet;
sum.inch = sum.inch - 12;
}

printf("Sum of distances = %d\'-%.1f\"", sum.feet, sum.inch);
return 0;
}
```

14.3 NESTED STRUCTURES

When a structure contains another structure, it is called nested structure. For example, we have two structures named Address and Employee. To make Address nested to Employee, we have to define Address structure before and outside Employee structure and create an object of Address structure inside Employee structure.

Syntax for structure within structure or nested structure

```
struct name_of_structure1
{
- - - - - - - - - -
- - - - - - - - - -
};

struct name_of_structure2
{
- - - - - - - - - -
- - - - - - - - - -
struct name_of_structure1 var_name;
};
```

Example for structure within structure or nested structure

```
#include<stdio.h>

struct Address
{
char area[20];
char town[25];
char pin[6];
};

struct Employee
{
int Id;
char Name[25];
```

```
    float salary;
    struct Address addr;
};

void main()
{
int i;
struct Employee emp1;

printf("\n\tEnter Employee Id : ");
scanf("%d",&emp1.Id);

printf("\n\tEnter Employee Name : ");
scanf("%s",&emp1.Name);

printf("\n\tEnter Employee salary : ");
scanf("%f",&emp1.salary);

printf("\n\tEnter area of residence : ");
scanf("%s",&emp1.addr.area);

printf("\n\tEnter Employee town : ");
scanf("%s",&emp1.addr.town);

printf("\n\tEnter Employee area : ");
scanf("%s",&emp1.addr.pin);

printf("\nDetails of Employees");
printf("\n\tEmployee Id : %d",emp1.Id);
printf("\n\tEmployee Name : %s",emp1.Name);
printf("\n\tEmployee salary : %f",emp1.salary);
printf("\n\tEmployee House area : %s",emp1.addr.area);
printf("\n\tEmployee town : %s",emp1.addr.town);
printf("\n\tEmployee pin Code : %s",emp1.addr.pin);
}
```

Output :

```
Enter Employee Id : 101
Enter Employee Name : Ajit Sharma
Enter Employee salary : 45000
Enter Employee area : Andheri
Enter Employee town : Mumbai
Enter Employee pin Code : 400033

Details of Employees
Employee Id : 101
Employee Name : Ajit Sharma
Employee salary : 45000
```

Employee area : Andheri
Employee town : Mumbai
Employee pin Code : 400033

14.5 STRUCTURES AND FUNCTIONS

We can pass a structure as a function argument in the same way as you pass any other variable or pointer. Let us revisit the sum of distances program and use a function in the program to add the 2 structure variables dist1 and dist 2

```c
// Program to add two distances (feet-inch) using functions
#include <stdio.h>

struct Distance
{
int feet;
float inch;
} dist1, dist2, sum;

void printSum( Struct Distance d1, Struct Distance d2)
{

sum.feet = d1.feet + dist2.feet;
sum.inch = d1.inch + dist2.inch;

// changing to feet if inch is greater than 12
while (sum.inch >= 12)
{
++sum.feet;
sum.inch = sum.inch - 12;
}

printf("Sum of distances = %d\'-%.1f\"", sum.feet, sum.inch);
}
int main()
{

printf("1st distance\n");
printf("Enter feet: ");
scanf("%d", &dist1.feet);

printf("Enter inch: ");
scanf("%f", &dist1.inch);
printf("2nd distance\n");

printf("Enter feet: ");
scanf("%d", &dist2.feet);
```

```
printf("Enter    inch:    ");
scanf("%f", &dist2.inch);

printSum( dist1 , dist2 );
printSum( dist1 , dist1 );
printSum( dist2 , dist2 );

return 0;
}
```

We print the sum of dist1 and dist2 , dist1 with itself and dist2 with itself in the above code, by calling the function printSum(dist1 , dist2) , printSum(dist1 , dist1) and printSum(dist2 , dist2) respectively.
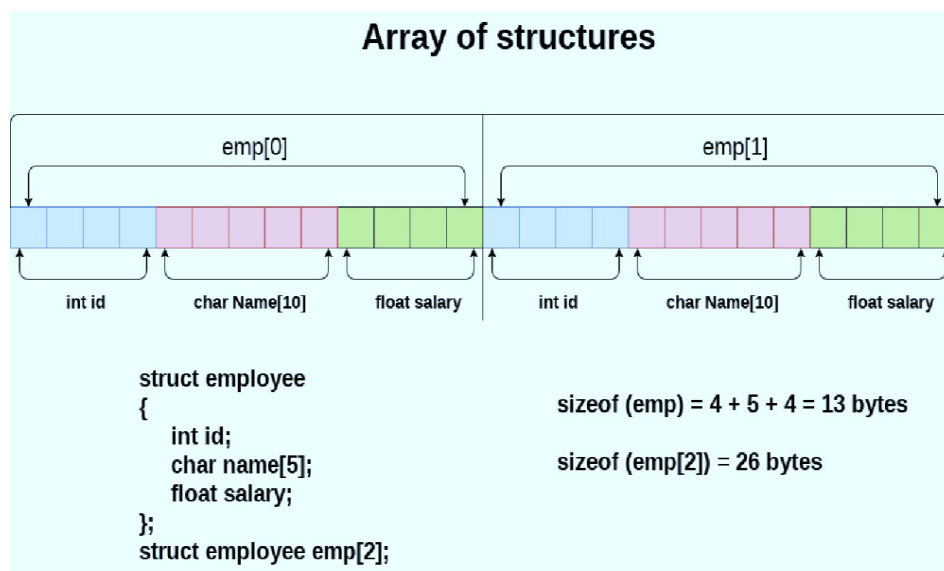
14.6 STRUCTURES AND ARRAYS

When we talk about structures and arrays let us look at the 2 ways they can be used
1. Arrays of structures
2. Structures containing arrays

We shall now consider each case briefly.

Arrays of Structures:
An array of structres in C can be defined as the collection of multiple structures variables where each variable contains information about different entities. The array of structures in C are used to store information about multiple entities of different data types. The array of structures is also known as the collection of structures.

Array of structures

emp[0]   emp[1]

int id   char Name[10]   float salary   int id   char Name[10]   float salary

```
struct employee
{
    int id;
    char name[5];
    float salary;
};
struct employee emp[2];
```

sizeof (emp) = 4 + 5 + 4 = 13 bytes

sizeof (emp[2]) = 26 bytes

Below is the demonstration of a program that uses the concept of the array within a structure.

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

struct student
{
int seatno;
float percentage;
};

int main()
{
int i;
struct student record[2];

// 1st student's record
record[0].seatno=1;
record[0].percentage = 86.5;

// 2nd student's record
record[1].seatno=2;
record[1].percentage = 90.5;

// 3rd student's record
record[2].seatno=3;
record[2].percentage = 81.5;

for(i=0; i<3; i++)
{
printf(" \n Records of STUDENT : %d \n", i+1);
printf(" Seat No is: %d \n", record[i].seatno);
printf(" Percentage is: %f\n\n",record[i].percentage);
}
return 0;
}
```

Output:

```
Records of STUDENT : 1
Seat No is: 1
Percentage is: 86.500000
Records of STUDENT : 2
Seat No is: 2
Percentage is: 90.500000
Records of STUDENT : 3
Seat No is: 3
Percentage is: 81.50000
```

Structures containing arrays:

Let us know discuss the use of arrays in a structure. For this we will see the example below which demonstrates the use of a structure called Books which contains 2 character arrays for storing the title and author of the book in addition to a integer variable for bookid.

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

struct Book {
int bk_id;
char title[50];
char author[50];

};

int main( ) {

struct Book bk1; /* Declare bk1 of type Book */
struct Book bk2; /* Declare bk2 of type Book */

/* book 1 specification */
bk1.bk_id = 650;
strcpy( bk1.title, "Let Us C");
strcpy( bk1.author, "Yashvant K");

/* book 2 specification */
bk1.bk_id = 651;
strcpy( bk2.title, "The Secret of Nagas");
strcpy( bk2.author, "Amish T");

x

/* print bk1 info */
printf( "Book 1 bk_id : %d\n", bk1.bk_id);
printf( "Book 1 title : %s\n", bk1.title);
printf( "Book 1 author : %s\n", bk1.author);

/* print bk2 info */
printf( "Book 2 bk_id : %d\n", bk2.bk_id);
printf( "Book 2 title : %s\n", bk2.title);
printf( "Book 2 author : %s\n", bk2.author);

return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
Book 1 bk_id : 650
Book 1 title : Let Us C
Book 1 author : Yashvant K
Book 2 bk_id : 651
Book 2 title : The Secret of Nagas
Book 2 author : Amish T
```

14.8 STRUCTURES AND POINTERS

You can define pointers to structures in the same way as you define pointer to any other variable −

struct Book *struct_pointer;

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the '&'; operator before the structure's name as follows −

struct_pointer = &Book1;

To access the members of a structure using a pointer to that structure, you must use the → operator as follows −

struct_pointer->title;

Let us re-write the above example using structure pointer.

Example:

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

struct Book {
char title[50];
int bk_id;
};

/* function declaration */
void printBook( struct Book *book );
int main( ) {

struct Book bk1, bk2

strcpy( bk1.title, "Let Us C");
bk1.bk_id = 750;

strcpy( bk2.title, "The Secret of Nagas");
```

```
bk2.bk_id = 751;

/*
/* print Book1 info by passing address of Book1 */
printBook( &bk1 );

/* print Book2 info by passing address of Book2 */
printBook( &bk2 );

return 0;
}

void printBook( struct Book *book ) {

printf( "Book title : %s\n", book->title);
printf( "Book bk_id : %d\n", book->bk_id);
}
```

When the above code is compiled and executed, it produces the following result
Book title   : Let Us C
Book bk_id : 750
Book title   : The Secret of Nagas
Book bk_id : 751

14.9 UNION

C Union is also like structure, i.e. collection of different data types which are grouped together. Each element in a union is called member.

Union and structure in C are same in concepts, except allocating memory for their members. Structure allocates storage space for all its members separately. Whereas, Union allocates one common storage space for all its members.

We can access only one member of union at a time. We can't access all member values at the same time in union. But, structure can access all member values at the same time. This is because, Union allocates one common storage space for all its members. Where as Structure allocates storage space for all its members separately.

Many union variables can be created in a program and memory will be allocated for each union variable separately.

Example:

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

```c
{
char name[20];
char subject[20];
float percentage;
};
int main()
{

union student record1;
union student record2;

// assigning values to record1 union variable
strcpy(record1.name, "Mayur");
strcpy(record1.subject, "Maths");
record1.percentage = 86.50;

printf("Union record1 values example\n");
printf(" Name : %s \n", record1.name);
printf(" Subject : %s \n", record1.subject);
printf(" Percentage : %f \n\n", record1.percentage);

// assigning values to record2 union variable
printf("Union record2 values example\n");
strcpy(record2.name, "Kiran");
printf(" Name : %s \n", record2.name);

strcpy(record2.subject, "Physics");
printf(" Subject : %s \n", record2.subject);

record2.percentage = 14.50;
printf(" Percentage : %f \n", record2.percentage);
return 0;
}
```

Output:

```
Union record1 values example
Name : Mayur
Subject : Maths
Percentage : 86.500000;
Union record2 values example
Name : Kiran
Subject : Physics
Percentage : 14.500000
```

14.10 UNIT END QUESTIONS

1. How do you declare a structure in C?

2. Create a data structure to store data about a music album. Create 3 instances of the album and display their values.

3. How do we declare a structure containing arrays? Give example.

4. Declare an array of structure called students. Initialize 5 students using inputs from user.

5. What are unions? when are they used in C ?

*****