

## Imperative Programming Syllabus

Unit	Details
I	<p><b>Introduction:</b> Types of Programming languages, History, features and application. Simple program logic, program development cycle, pseudocode statements and flowchart symbols, sentinel value to end a program, programming and user environments, evolution of programming models., desirable program characteristics.</p> <p><b>Fundamentals:</b> Structure of a program. Compilation and Execution of a Program, Character Set, identifiers and keywords, data types, constants, variables and arrays, declarations, expressions, statements, Variable definition, symbolic constants.</p>
II	<p><b>Operators and Expressions:</b> Arithmetic operators, unary operators, relational and logical operators, assignment operators, assignment operators, the conditional operator, library functions.</p> <p><b>Data Input and output:</b> Single character input and output, entering input data, scanf function, printf function, gets and puts functions, interactive programming.</p>
III	<p><b>Conditional Statements and Loops:</b> Decision Making Within A Program, Conditions, Relational Operators, Logical Connectives, If Statement, If-Else Statement, Loops: While Loop, Do While, For Loop. Nested Loops, Infinite Loops, Switch Statement</p> <p><b>Functions:</b> Overview, defining a function, accessing a function, passing arguments to a function, specifying argument data types, function prototypes, recursion, modular programming and functions, standard library of c functions, prototype of a function: foo l l a l parameter list, return type, function call, block structure, passing arguments to a function: call by reference, call by value.</p>
IV	<p><b>Program structure:</b> Storage classes, automatic variables, external variables, static variables, multifile programs, more library functions,</p> <p><b>Preprocessor:</b> Features, #define and #include, Directives and Macros</p> <p><b>Arrays:</b> Definition, processing, passing arrays to functions, multidimensional arrays, arrays and strings.</p>
V	<p><b>Pointers:</b> Fundamentals, declarations, Pointers Address Operators, Pointer Type Declaration, Pointer Assignment, Pointer Initialization, Pointer Arithmetic, Functions and Pointers, Arrays And Pointers, Pointer Arrays, passing functions to other functions</p> <p><b>Structures and Unions:</b> Structure Variables, Initialization, Structure Assignment,</p>

	Nested Structure, Structures and Functions, Structures and Arrays: Arrays of Structures, Structures Containing Arrays, Unions, Structures and pointers.
--	---

## **UNIT 1: CHAPTER -1**

**Introduction:** Types of Programming languages, History, features and application. Simple program logic, program development cycle, pseudocode statements and flowchart symbols, sentinel value to end a program, programming and user environments, evolution of programming models., desirable program characteristics.

### **[1.1] TYPES OF PROGRAMMING LANGUAGES:-**

There are many different languages can be used to program a computer. The most basic of these is machine language--a collection of very detailed, cryptic instructions that control the computer's internal circuitry. This is the natural dialect of the computer. Very few computer programs are actually written in machine language, however, for two significant reasons: 1)First, because machine language is very cumbersome to work with and

2)second, because every different type of computer has its own unique instruction set.

Thus, a machine-language program written for one type of computer cannot be run on another type of computer without significant alterations. Usually, a computer program will be written in some high-level language, whose instruction set is more compatible with human languages and human thought processes. Most of these are general-purpose languages such as C. (Some other popular general-purpose languages are Pascal, Fortran and BASIC.) There are also various special-purpose languages that are specifically designed for some particular type of application. Some common examples are CSMP and SIMAN, which are special-purpose simulation languages, and LISP, a list-processing language that is widely used for artificial intelligence applications.

As a rule, a single instruction in a high-level language will be equivalent to several instructions in machine language. This greatly simplifies the task of writing complete, correct programs. Furthermore, the 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. Thus, we see that a high-level language offers three significant advantages over machine language: simplicity, uniformity and portability (i.e., machine independence).

A program that is written in a high-level language must, however, 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.) In either case, the translation is carried out automatically within the computer. In fact, inexperienced programmers may not even be aware that this process is taking place, since they typically see only their original high-level

program, the input data, and the calculated results. Most implementations of C operate as compilers.

A compiler or interpreter is itself a computer program. It accepts a program written in a high-level language (e.g., C) as input, 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 computer must have its own compiler or interpreter for a particular high-level language. 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, however, a compiled version will normally execute much faster than an interpreted version. The reasons for this are beyond the scope of our present discussion.

### **INTRODUCTION TO C:-**

C is a general-purpose, structured programming language. Its instructions consist of terms that resemble algebraic expressions, augmented by certain English keywords such as if, else, for, do and while. In this respect C resembles other high-level structured programming languages such as Pascal and Fortran. C also contains certain additional features, however, that allow it to be used at a lower level, thus bridging the gap between machine language and the more conventional high-level languages. This flexibility allows C to be used for systems programming (e.g., for writing operating systems) as well as for applications programming (e.g., for writing a program to solve a complicated system of mathematical equations, or for writing a program to bill customers).

C is characterized by the ability to write very concise source programs, due in part to the large number of operators included within the language. It has a relatively small instruction set, though actual implementations include extensive library functions which enhance the basic instructions. Furthermore, the language encourages users to write additional library functions of their own. Thus the features and capabilities of the language can easily be extended by the user.

C compilers are commonly available for computers of all sizes, and C interpreters are becoming increasingly common. The compilers are usually compact, and they generate object programs that are small and highly efficient when compared with programs compiled from other high-level languages. The interpreters are less efficient, though they are easier to use when developing a new program. Many programmers begin with an interpreter, and then switch to a compiler once the program has been debugged (i.e., once all of the programming errors have been removed).

Another important characteristic of C is that its programs are highly portable, even more so than with other high-level languages. The reason for this is that C relegates most computer-dependent features to its library functions. Thus, every version of C is accompanied by its own set of library functions, which are written for the particular

characteristics of the host computer. These library functions are relatively standardized, however, and each individual library function is generally accessed in the same manner from one version of C to another. Therefore, most C programs can be processed on many different computers with little or no alteration.

**It can be defined by the following ways:**

1. Mother language
2. System programming language
3. Procedure-oriented programming language
4. Structured programming language
5. Mid-level programming language

## **[1.2] HISTORY OF C:**

C was originally developed in the 1970s by Dennis Ritchie at Bell Telephone Laboratories, Inc. (now a part of AT&T). It is an outgrowth of two earlier languages, called BCPL and B, which were also developed at Bell Laboratories. C was largely confined to use within Bell Laboratories until 1978, when Brian Kernighan and Ritchie published a definitive description of the language.\* The Kernighan and Ritchie description is commonly referred to as "K&R C."

Following the publication of the K&R description, computer professionals, impressed with C's many desirable features, began to promote the use of the language. By the mid 1980s, the popularity of C had become widespread. Numerous C compilers and interpreters had been written for computers of all sizes, and many commercial application programs had been developed. Moreover, many commercial software products that were originally written in other languages were rewritten in C in order to take advantage of its efficiency and its portability.

Early commercial implementations of C differed somewhat from Kernighan and Ritchie's original definition, resulting in minor incompatibilities between different implementations of the language. These differences diminished the portability that the language attempted to provide. Consequently, the American National Standards Institute\*\* (ANSI committee X3J11) has developed a standardized definition of the C language. Virtually all commercial C compilers and interpreters now adhere to the ANSI standard. Many also provide additional features of their own. In the early 1980s, another high-level programming language, called C++, was developed by Bjarne Stroustrup\*\*\* at the Bell Laboratories. C++ is built upon C, and hence all standard C features are available within C++. However, C++ is not merely an extension of C. Rather, it incorporates several new fundamental concepts that form a basis for object-oriented programming--a new programming paradigm that is of interest to professional

programmers. We will not describe C++ in this book, except to mention that a knowledge of C is an excellent starting point for learning C++.

This book describes the features of C that are included in the ANSI standard and are supported by commercial C compilers and interpreters. The reader who has mastered this material should have no difficulty in customizing a C program to any particular implementation of the language.

### [1.3]FEATURES AND APPLICATION:-

#### 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**.

#### Application of C programming:-

1. C language is used for creating **computer applications**
2. Used in writing **Embedded softwares**
3. Firmware for various electronics, industrial and communications products which use micro-controllers.
4. It is also used in developing **verification software, test code, simulators** etc. for various applications and hardware products.
5. **For Creating Compiles** of different Languages which can take input from other language and convert it into lower level machine dependent language.
6. C is used to implement different **Operating System Operations**.
7. **UNIX kernel** is completely developed in C Language.

#### List of Applications of C Programming

---

List of Application		
Operating Systems	Network Drivers	Print Spoolers
Language Compilers	Assemblers	Text Editors
Modern Programs	Data Bases	Language Interpreters
Simulators	Utilities	Embedded System

## [1.4] SIMPLE PROGRAM LOGIC:-

**Code:-**

---

```
//C hello world example
#include <stdio.h>

int main()
{
    printf("Hello world\n");
    return 0;
}
```

**Output:- Hello world**

## [1.5] PROGRAM DEVELOPMENT CYCLE:-

### C and C++ Program Development Cycle

A C or C++ program typically passes through four steps of development.

**Design and Code** Involves designing a program to meet a specified requirement, and creating the programming language text files that will comprise the program source.

**Compile** After checking for syntactical correctness, converts the programming language source files into machine readable instructions, where C or C++ variables are associated with memory addresses, and C or C++ statements are turned into a series of machine language instructions. The compiler can produce various forms of output, depending on the compiler options selected.

**Linkage Editor** Links compiler output with external modules requested by the compiled program. Programs can use routines from C and C++ libraries or any object or archive file from the IBM XL family of languages. Programs can also use modules produced by the current or previous compilations. As well as linking the external modules, the linkage editor resolves addresses within the object module.

**Run and** This stage can be both the final step in program development, or it can be an



**Test** intermediate point in the program design and implementation process. A program's design commonly is further refined as a result of information gathered during testing.

## [1.6] PSEUDOCODE STATEMENTS AND FLOWCHART SYMBOLS

### Pseudocode and Flowcharts

Good, logical programming is developed through good pre-code planning and organization. This is assisted by the use of pseudocode and program flowcharts.

**Flowcharts** are written with program flow from the top of a page to the bottom. Each command is placed in a box of the appropriate shape, and arrows are used to direct program flow. The following shapes are often used in flowcharts:



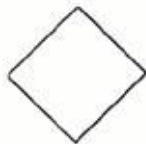
An oval indicates beginning or end of a program.



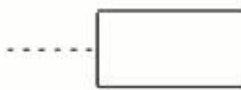
A parallelogram is a point where there is input to or output from the program.



A rectangle indicates the assignment of a value to a variable, constant, or parameter. the assigned value can be the result of a computation. The computation would also be included in the rectangle.



A diamond indicates a point where a decision is made.



An open-ended rectangle contains comment statements. The comment is connected to the program flow via a dashed line.



A hexagon indicates the beginning of a repetition.



The double-lined rectangle indicates the use of an algorithm specified outside the program, such as a subroutine.



Circles can be used to combine flow lines.

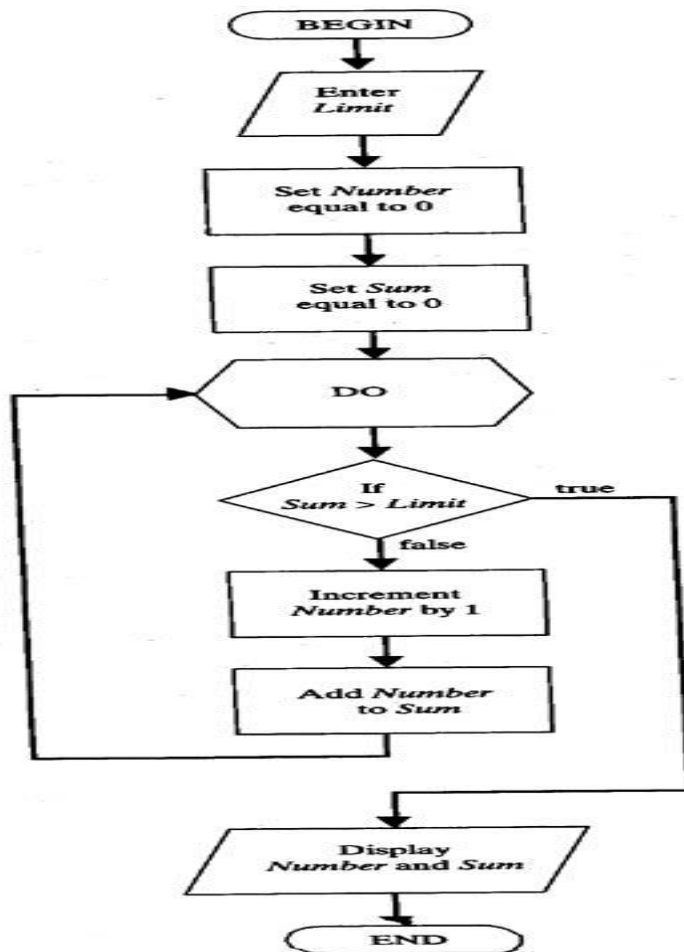


Arrows indicate the direction and order of program execution.

**Pseudocode** is a method of describing computer algorithms using a combination of natural language and programming language. It is essentially an intermittent step towards the development of the actual code. It allows the programmer to formulate their thoughts on the organization and sequence of a computer algorithm without the need for actually following the exact coding syntax. Although pseudocode is frequently used there are no set of rules for its exact implementation. In general, here are some rules that are frequently followed when writing pseudocode:

- The usual Fortran symbols are used for arithmetic operations (+, -, \*, /, \*\*).
- Symbolic names are used to indicate the quantities being processed.
- Certain Fortran keywords can be used, such as PRINT, WRITE, READ, etc.
- Indentation should be used to indicate branches and loops of instruction.
- Here is an example problem, including a flowchart, pseudocode,
- For a given value, *Limit*, what is the smallest positive integer *Number* for which the sum  
 $Sum = 1 + 2 + \dots + Number$  is greater than *Limit*. What is the value for this *Sum*?
- Pseudocode:
  - Input: An integer *Limit*
  - Output: Two integers: *Number* and *Sum*
- 1. Enter *Limit*
  2. Set *Number* = 0.
  3. Set *Sum* = 0.
  4. Repeat the following:
    - a. If  $Sum > Limit$ , terminate the repetition, otherwise.
    - b. Increment *Number* by one.
    - c. Add *Number* to *Sum* and set equal to *Sum*.
  5. Print *Number* and *Sum*.

**Flowchart:**



### Code:

- PROGRAM Summation
- ! Program to find the smallest positive integer Number  
! For which  $\text{Sum} = 1 + 2 + \dots + \text{Number}$   
! is greater than a user input value Limit.
- IMPLICIT NONE
- ! Declare variable names and types
- INTEGER :: Number, Sum, Limit
- ! Initialize Sum and Number
- Number = 0
- Sum = 0
- ! Ask the user to input Limit
- PRINT \*, "Enter the value for which the sum is to exceed:"
- READ \*, Limit
- ! Create loop that repeats until the smallest value for Number is found.
- DO
  - IF (Sum > Limit) EXIT ! Terminate repetition once Number is found
  - ! otherwise increment number by one
  - Number = Number + 1
  - Sum = Sum + 1
- END DO
- ! Print the results

- PRINT \*, "1 + ... + ", Number, "=", Sum, ">", Limit
- END PROGRAM

### [1.7] SENTINEL VALUE TO END A PROGRAM:-

The program can be used to add up a list of numbers. But the user first has to count up how many numbers are to be added up. This could be annoying. It would be nice to have a program that keeps adding up numbers until you tell it to stop. Here is a program that does that. The user will enter numbers, one by one, and the program will add each number to SUM. The user tells the program to stop by entering a zero. The zero is how the user "signals" that the program should stop looping.

A sentinel value is a special value that is tested in a DO WHILE statement to determine if the loop should end.

The program is not quite finished, yet.

Add up numbers that the user enters.

When the user enters 0, print the sum and end the program.

```
,
LET SUM = 0
,
PRINT "Enter a number"
INPUT NUMBER
,
DO WHILE NUMBER ____ 0
    LET SUM = SUM + NUMBER
    PRINT "Enter a number. (Type 0 to stop)"
    INPUT NUMBER
LOOP
,
PRINT "The sum is", SUM
END
```

The DO WHILE statement acts like a gatekeeper. It allows execution to (re-)enter the loop body only if the condition is true. Say that the program has just started. The user wants to add up a list of numbers. The first number is 4, so the user enters it:

**Output:-**

**Enter a number**

? 4

Now the program should execute the first statement in the loop body.

### **[1.8] DESIRABLE PROGRAM CHARACTERISTICS:-**

Before concluding this chapter let us briefly examine some important characteristics of well-written computer programs. These characteristics apply to programs that are written in any programming language, not just C. They can provide us with a useful set of guidelines later in this book, when we start writing our own C programs.

1. Integrity. This refers to the accuracy of the calculations. It should be clear that all other program enhancements will be meaningless if the calculations are not carried out correctly. 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 particular emphasis on its underlying logic. If a program is clearly written, it should be possible for another programmer to follow the program logic without undue 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 programs through an orderly and disciplined approach to programming.

3. Simplicity. The clarity and accuracy of a program are usually enhanced by keeping things as simple as possible, consistent 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 is concerned with execution speed and efficient memory utilization. These are generally important goals, though they should not be obtained at the expense of clarity or simplicity. Many complex programs require a tradeoff between these characteristics. In such situations, experience and common sense are key factors.

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, 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.Generalitv. Usually we will want a 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 generalitv can be obtained with very little additional programming effort.

## **UNIT 1: CHAPTER 2**

**Fundamentals:** Structure of a program. Compilation and Execution of a Program, Character Set, identifiers and keywords, data types, constants, variables and arrays, declarations, expressions, statements, Variable definition, symbolic constants.

### **[2.1]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 (more about this later, in Chaps. 7 and 8).

#### **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., /\* this 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.

**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) * I

scanf ( "%f", &radius) ; /* INPUT STATEMENT */

area = 3.14159 * radius * radius; /* ASSIGNMENT STATEMENT */

printf ("Area = %f", area) ; /* OUTPUT STATEMENT */

1

```

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 circle */

#include <stdio.h>

main( )

float radius, area;

printf ("Radius = ? ");

scanf ("%f", &radius) ;

area = 3.14159 * radius * radius;

printf ("Area = %f", area) ;

```

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

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.

(Uppercase and lowercase characters are not equivalent in C. Later in this book we will see some special situations that are characteristically typed in uppercase.)

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 (`printf`) 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 (`printf`) 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.
11. 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.

Output:-

**Radius = 7 3**

**Area = 28.274309**

## [2.2] COMPILATION AND EXECUTION OF A PROGRAM:-

Whenever a C program file is compiled and executed, the compiler generates some files with the same name as that of the C program file but with different extensions. So, what are these files and how are they created?

Below image shows the compilation process with the files created at each step of the compilation process:

Every file that contains a C program must be saved with '.c' extension. This is necessary for the compiler to understand that this is a C program file. Suppose a program file is named, first.c. The file first.c is called the source file which keeps the code of the program. Now, when we compile the file, the C compiler looks for errors. If the C compiler reports no error, then it stores the file as a .obj file of the same name, called the object file. So, here it will create the first.obj. This .obj file is not executable. The process is continued by the Linker which finally gives a .exe file which is executable.

**Linker:** First of all, let us know that library functions are not a part of any C program but of the C software. Thus, the compiler doesn't know the operation of any function, whether it be printf or scanf. The definitions of these functions are stored in their respective library which the compiler should be able to link. This is what the Linker does. So, when we write #include, it includes stdio.h library which gives access to Standard Input and Output. The linker links the object files to the library functions and the program becomes a .exe file. Here, first.exe will be created which is in an executable format.

**Loader:** Whenever we give the command to execute a particular program, the loader comes into work. The loader will load the .exe file in RAM and inform the CPU with the starting point of the address where this program is loaded.

**Instruction Register:** It holds the current instructions to be executed by the CPU.

**Program Counter:** It contains the address of the next instructions to be executed by the CPU.

**Accumulator:** It stores the information related to calculations.

The loader informs Program Counter about the first instruction and initiates the execution. Then onwards, Program Counter handles the task.

## [2.3] CHARACTER SET

As every language contains a set of characters used to construct words, statements, etc., C language also has a set of characters which include **alphabets**, **digits**, and **special symbols**. C language supports a total of 256 characters.

Every C program contains statements. These statements are constructed using words and these words are constructed using characters from C character set. C language character set contains the following set of characters...

1. Alphabets
2. Digits
3. Special Symbols

## Alphabets

C language supports all the alphabets from the English language. Lower and upper case letters together support 52 alphabets.

lower case letters - **a to z**

UPPER CASE LETTERS - **A to Z**

## Digits

C language supports 10 digits which are used to construct numerical values in C language.

Digits - **0, 1, 2, 3, 4, 5, 6, 7, 8, 9**

## Special Symbols

C language supports a rich set of special symbols that include symbols to perform mathematical operations, to check conditions, white spaces, backspaces, and other special symbols.

Special Symbols - **~ @ # \$ % ^ & \* ( ) \_ - + = { } [ ] ; : ' " / ? . > , < \ | tab newline space NULL bell backspace verticaltab etc.,**

Every character in C language has its equivalent ASCII (American Standard Code for Information Interchange) value

### C program to print all the characters of C character Set

```
#include<stdio.h>
#include<conio.h>
int main() {
    int i;
    clrscr();
```

```
printf("ASCII ==> Character\n");
for(i = -128; i <= 127; i++)
    printf("%d ==> %c\n", i, i);
getch();
return 0;
}
```

## [2.4] IDENTIFIERS AND KEYWORDS:

### IDENTIFIERS:-

In C programming language, programmers can specify their name to a variable, array, pointer, function, etc... An identifier is a collection of characters which acts as the name of variable, function, array, pointer, structure, etc... In other words, an identifier can be defined as the user-defined name to identify an entity uniquely in the c programming language that name may be of the variable name, function name, array name, pointer name, structure name or a label.

**The identifier is a user-defined name of an entity to identify it uniquely during the program execution**

### Example

```
int marks;
char studentName[30];
```

Here, **marks** and **studentName** are identifiers.

### Rules for Creating Identifiers

1. An identifier can contain **letters** (UPPERCASE and lowercase), **numerics** & **underscore** symbol only.
2. An identifier should not start with a numerical value. It can start with a letter or an underscore.
3. We should not use any special symbols in between the identifier even whitespace. However, the only underscore symbol is allowed.
4. Keywords should not be used as identifiers.
5. There is no limit for the length of an identifier. However, the compiler considers the first 31 characters only.
6. An identifier must be unique in its scope.

### Rules for Creating Identifiers for better programming

The following are the commonly used rules for creating identifiers for better programming...

1. The identifier must be meaningful to describe the entity.
2. Since starting with an underscore may create conflict with system names, so we avoid starting an identifier with an underscore.
3. We start every identifier with a lowercase letter. If an identifier contains more than one word then the first word starts with a lowercase letter and second word onwards first letter is used as an UPPERCASE letter. We can also use an underscore to separate multiple words in an identifier.

## KEYWORDS:-

As every language has words to construct statements, C programming also has words with a specific meaning which are used to construct c program instructions. In the C programming language, keywords are special words with predefined meaning. Keywords are also known as reserved words in C programming language.

In the C programming language, there are **32 keywords**. All the 32 keywords have their meaning which is already known to the compiler.

**Keywords are the reserved words with predefined meaning which already known to the compiler**

Whenever C compiler come across a keyword, automatically it understands its meaning.

## Properties of Keywords

1. All the keywords in C programming language are defined as lowercase letters so they must be used only in lowercase letters
2. Every keyword has a specific meaning, users can not change that meaning.
3. Keywords can not be used as user-defined names like variable, functions, arrays, pointers, etc...
4. Every keyword in C programming language represents something or specifies some kind of action to be performed by the compiler.

The following table specifies all the 32 keywords

### KEYWORDS

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

## [2.5] DATA TYPES

Each variable in C has an associated data type. Each data type requires different amounts of memory and has some specific operations which can be performed over it. Let us briefly describe them one by one:

Following are the examples of some very common data types used in C:

- **char:** The most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.
- **int:** As the name suggests, an int variable is used to store an integer.
- **float:** It is used to store decimal numbers (numbers with floating point value) with single precision.
- **double:** It is used to store decimal numbers (numbers with floating point value) with double precision.

Different data types also have different ranges upto which they can store numbers. These ranges may vary from compiler to compiler. Below is list of ranges along with the memory requirement and format specifiers on 32 bit gcc compiler.

DATA TYPE	MEMORY (BYTES)	RANGE	FORMAT SPECIFIER
short int	2	-32,768 to 32,767	%hd
unsigned short int	2	0 to 65,535	%hu
unsigned int	4	0 to 4,294,967,295	%u
int	4	-2,147,483,648 to 2,147,483,647	%d
long int	8	-2,147,483,648 to 2,147,483,647	%ld

DATA TYPE	MEMORY (BYTES)	RANGE	FORMAT SPECIFIER
unsigned long int	8	0 to 4,294,967,295	%lu
long long int	8	-(2^63) to (2^63)-1	%lld
unsigned long long int	8	0 to 18,446,744,073,709,551,615	%llu
signed char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
float	4		%f
double	8		%lf
long double	16		%Lf

## [2.6[CONSTANTS, VARIABLES AND ARRAYS, DECLARATIONS, EXPRESSIONS, STATEMENTS, VARIABLE DEFINITION, SYMBOLIC CONSTANTS

### a)CONSTANTS:

A constant is a value or variable that can't be changed in the program, for example: 10, 20, 'a', 3.4, "c programming" etc.

There are different types of constants in C programming.

#### List of Constants in C

Constant	Example
Decimal Constant	10, 20, 450 etc.
Real or Floating-point Constant	10.3, 20.2, 450.6 etc.
Octal Constant	021, 033, 046 etc.
Hexadecimal Constant	0x2a, 0x7b, 0xaa etc.
Character Constant	'a', 'b', 'x' etc.
String Constant	"c", "c program", "c in javatpoint" etc.

### b)VARIABLES:

A **variable** is a name of the memory location. It is used to store data. Its value can be changed, and it can be reused many times.

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.



It is a way to represent memory location through symbol so that it can be easily identified.

Let's see the syntax to declare a variable:

1. `type variable_list;`

The example of declaring the variable is given below:

1. `int a;`
2. `float b;`
3. `char c;`

Here, a, b, c are variables. The int, float, char are the data types.

We can also provide values while declaring the variables as given below:

1. `int a=10,b=20; //declaring 2 variable of integer type`
2. `float f=20.8;`
3. `char c='A';`

C programming language also allows to define various other types of variables, which we will cover in subsequent chapters like Enumeration, Pointer, Array, Structure, Union, etc. For this chapter, let us study only basic variable types.

### c)ARRAYS:-

Arrays are 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 –

```
type arrayName [ 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 –

```
balance[4] = 50.0;
```

The above statement assigns the 5<sup>th</sup> 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

## 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 10<sup>th</sup> 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 */
    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("Element[%d] = %d\n", j, n[j] );
}

return 0;
}

```

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

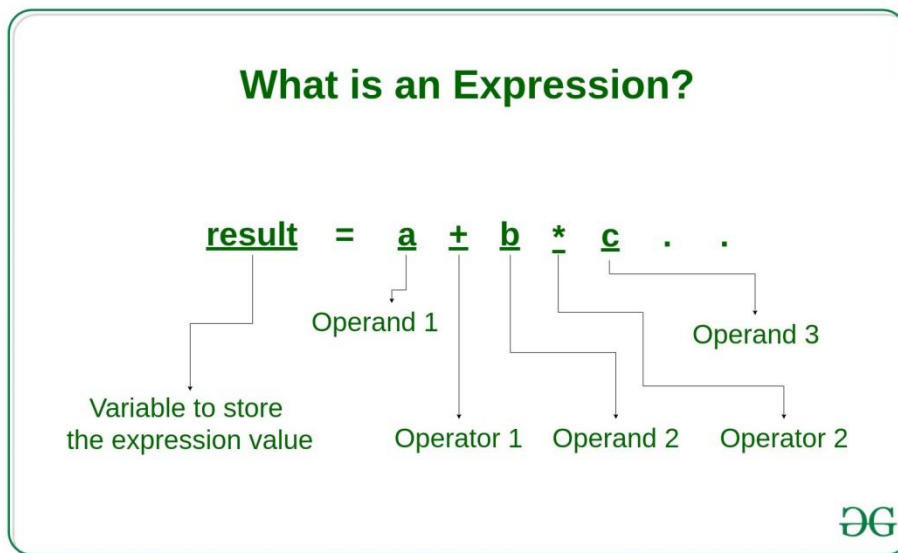
```

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

```

#### d)Expression:

An expression is a combination of operators, constants and variables. An expression may consist of one or more operands, and zero or more operators to produce a value.



#### e)SYMBOLIC CONSTANTS:

A symbolic constant is a name given to some numeric constant, or a character constant or string constant, or any other constants. Symbolic constant names are also known as constant identifiers. Pre-processor directive `#define` is used for defining symbolic constants.

## UNIT 2:CHAPTER -1

### **Operators and Expressions**

Arithmetic operators, unary operators, relational and logical operators, assignment operators, the conditional operator, library functions.

#### **OPERATORS:**

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.

#### **Arithmetic Operators:-**

The following table shows all the arithmetic operators supported by the C language. Assume variable **A** holds 10 and variable **B** holds 20 then –

Show Examples

Operator	Description	Example
+	Adds two operands.	$A + B = 30$
-	Subtracts second operand from the first.	$A - B = -10$
*	Multiplies both operands.	$A * B = 200$
/	Divides numerator by de-numerator.	$B / A = 2$

%	Modulus Operator and remainder of after an integer division.	B % A = 0
++	Increment operator increases the integer value by one.	A++ = 11
--	Decrement operator decreases the integer value by one.	A-- = 9

## Relational Operators

The following table shows all the relational operators supported by C. Assume variable **A** holds 10 and variable **B** holds 20 then –

Show Examples

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 –

Show Examples

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

$A \wedge B = 0011\ 0001$

$\sim A = 1100\ 0011$

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

[Show Examples](#)

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 –

[Show Examples](#)

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.

#### Show Examples

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

### 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.

Show Examples

Category	Operator	Associativity
Postfix	<code>() [] -&gt; . ++ --</code>	Left to right
Unary	<code>+ - ! ~ ++ -- (type)* &amp; sizeof</code>	Right to left
Multiplicative	<code>* / %</code>	Left to right
Additive	<code>+ -</code>	Left to right
Shift	<code>&lt;&lt; &gt;&gt;</code>	Left to right
Relational	<code>&lt; &lt;= &gt; &gt;=</code>	Left to right
Equality	<code>== !=</code>	Left to right
Bitwise AND	<code>&amp;</code>	Left to right
Bitwise XOR	<code>^</code>	Left to right
Bitwise OR	<code> </code>	Left to right
Logical AND	<code>&amp;&amp;</code>	Left to right
Logical OR	<code>  </code>	Left to right

Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^=  =	Right to left
Comma	,	Left to right

### CONDITIONAL OPERATOR IN C:-

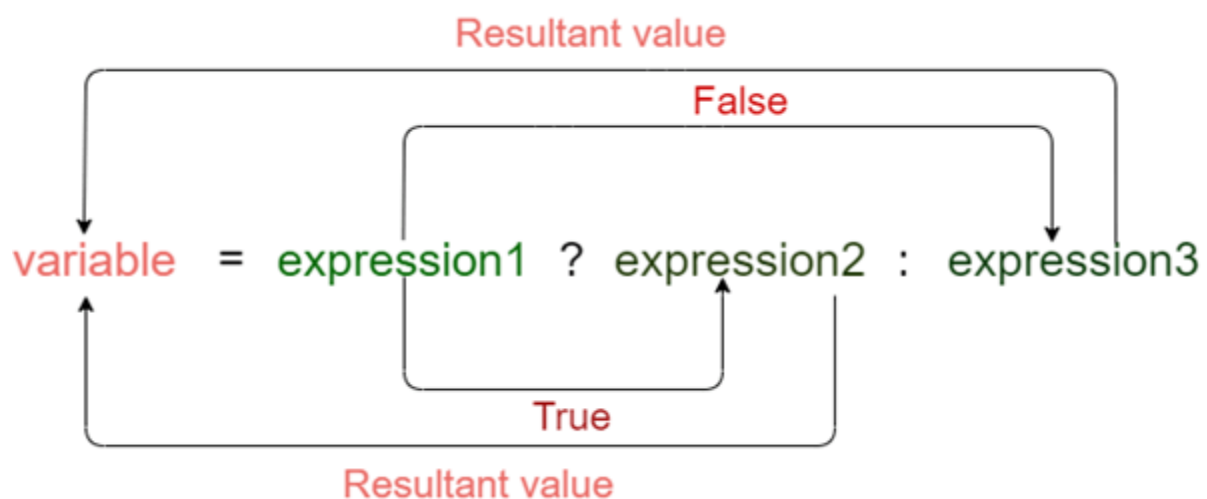
The conditional operator is also known as a **ternary operator**. The conditional statements are the decision-making statements which depends upon the output of the expression. It is represented by two symbols, i.e., '?' and ':'.

As conditional operator works on three operands, so it is also known as the ternary operator.

The behavior of the conditional operator is similar to the 'if-else' statement as 'if-else' statement is also a decision-making statement.

#### Syntax of a conditional operator

Expression1? expression2: expression3;



#### Meaning of the above syntax.

- In the above syntax, the expression1 is a Boolean condition that can be either true or false value.
- If the expression1 results into a true value, then the expression2 will execute.
- The expression2 is said to be true only when it returns a non-zero value.
- If the expression1 returns false value then the expression3 will execute.

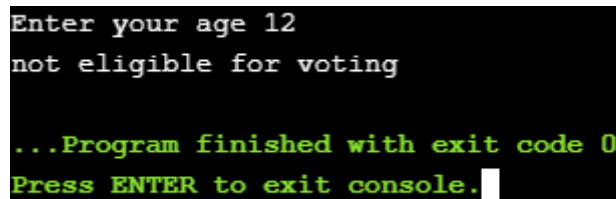
- The expression3 is said to be false only when it returns zero value.

Example:-

```
1. #include <stdio.h>
2. int main()
3. {
4.     int age; // variable declaration
5.     printf("Enter your age");
6.     scanf("%d",&age); // taking user input for age variable
7.     (age>=18)? (printf("eligible for voting")) : (printf("not eligible for voting")); // conditional operator
8.     return 0;
9. }
```

Output:-

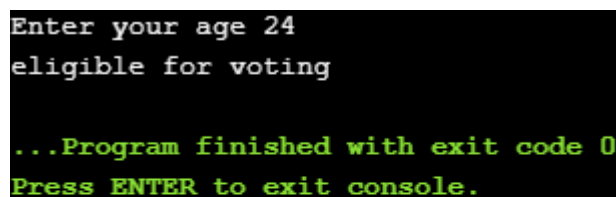
If we provide the age of user below 18, then the output would be:



```
Enter your age 12
not eligible for voting

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

If we provide the age of user above 18, then the output would be:



```
Enter your age 24
eligible for voting

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

In the above code, we are taking input as the 'age' of the user. After taking input, we have applied the condition by using a conditional operator. In this condition, we are checking the age of the user. If the age of the user is greater than or equal to 18, then the statement1 will execute, i.e., (printf("eligible for voting")) otherwise, statement2 will execute, i.e., (printf("not eligible for voting")).

## Library Functions:

The C library functions are provided by the system and stored in the library. The C library function is also called an inbuilt function in C programming.

To use Inbuilt Function in C, you must include their respective header files, which contain prototypes and data definitions of the function.

### Example:-

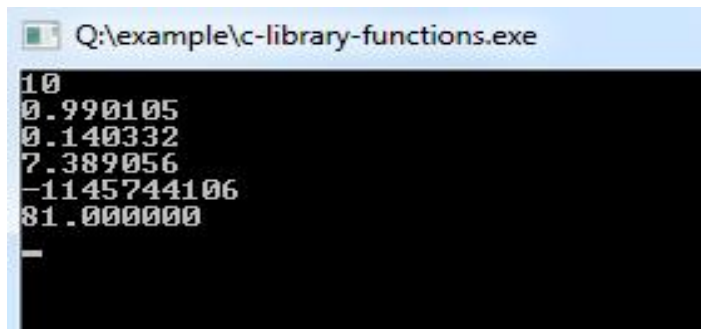
```
#include<stdio.h>

#include<ctype.h>

#include<math.h>

void main()
{
    int i = -10, e = 2, d = 10; /* Variables Defining and Assign values */ float rad = 1.43;
    double d1 = 3.0, d2 = 4.0;
    printf("%d\n", abs(i));
    printf("%f\n", sin(rad));
    printf("%f\n", cos(rad));
    printf("%f\n", exp(e));
    printf("%d\n", log(d));
    printf("%f\n", pow(d1, d2));
}
```

### Program Output:



```
Q:\example\c-library-functions.exe
10
0.990105
0.140332
7.389056
-1145744106
81.000000
_
```

## UNIT 2: CHAPTER -2

### Data Input and output

Single character input and output, entering input data, scanf function, printf function, gets and puts functions, interactive programming.

#### Single character input and output:-

When we say **Input**, it means to feed some data into a program. An input can be given in the form of a file or from the command line. C programming provides a set of built-in functions to read the given input and feed it to the program as per requirement.

When we say **Output**, it means to display some data on screen, printer, or in any file. C programming provides a set of built-in functions to output the data on the computer screen as well as to save it in text or binary files.

#### The Standard Files

C programming treats all the devices as files. So devices such as the display are addressed in the same way as files and the following three files are automatically opened when a program executes to provide access to the keyboard and screen.

Standard File	File Pointer	Device
Standard input	Stdin	Keyboard
Standard output	Stdout	Screen
Standard error	Stderr	Your screen

The file pointers are the means to access the file for reading and writing purpose. This section explains how to read values from the screen and how to print the result on the screen.

#### The `getchar()` and `putchar()` Functions:

The **int getchar(void)** function reads the next available character from the screen and returns it as an integer. This function reads only single character at a time. You can use this method in the loop in case you want to read more than one character from the screen.

The **int putchar(int c)** function puts the passed character on the screen and returns the same character. This function puts only single character at a time. You can use this method in the loop in case you want to display more than one character on the screen. Check the following example –

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

    int c;

    printf( "Enter a value :");
    c = getchar( );

    printf( "\nYou entered: ");
    putchar( c);

    return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads only a single character and displays it as follows –

```
./a.out
Enter a value : this is test
You entered: t
```

## The gets() and puts() Functions:-

The **char \*gets(char \*s)** function reads a line from **stdin** into the buffer pointed to by **s** until either a terminating newline or EOF (End of File).

The **int puts(const char \*s)** function writes the string 's' and 'a' trailing newline to **stdout**.

**NOTE:** Though it has been deprecated to use gets() function, Instead of using gets, you want to use fgets().

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

    char str[100];
```



```
printf( "Enter a value :");
gets( str );

printf( "\nYou entered: ");
puts( str );

return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads the complete line till end, and displays it as follows –

**Output:-**

**Enter a value :** this is test  
**You entered:** this is test

## The scanf() and printf() Functions:

The **int scanf(const char \*format, ...)** function reads the input from the standard input stream **stdin** and scans that input according to the **format** provided.

The **int printf(const char \*format, ...)** function writes the output to the standard output stream **stdout** and produces the output according to the format provided.

The **format** can be a simple constant string, but you can specify %s, %d, %c, %f, etc., to print or read strings, integer, character or float respectively. There are many other formatting options available which can be used based on requirements. Let us now proceed with a simple example to understand the concepts better –

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

    char str[100];
    int i;

    printf( "Enter a value :");
    scanf("%s %d", str, &i);

    printf( "\nYou entered: %s %d ", str, i);

    return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then program proceeds and reads the input and displays it as follows –

**Output:-**

**Enter a value :** seven 7

**You entered:** seven 7

Here, it should be noted that `scanf()` expects input in the same format as you provided `%s` and `%d`, which means you have to provide valid inputs like "string integer". If you provide "string string" or "integer integer", then it will be assumed as wrong input. Secondly, while reading a string, `scanf()` stops reading as soon as it encounters a space, so "this is test" are three strings for `scanf()`.

## **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.

## UNIT 3: CHAPTER-1

### **Conditional Statements and Loops:**

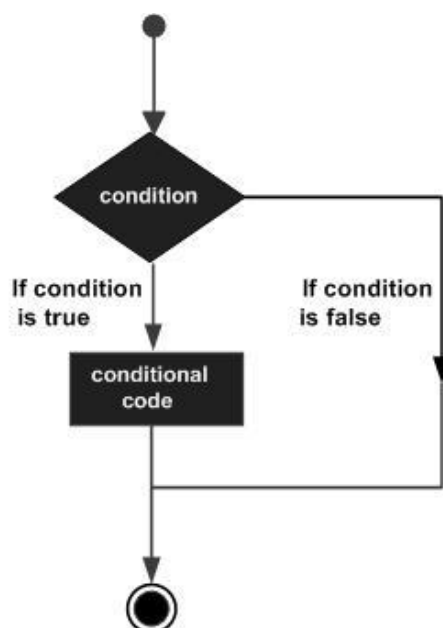
Decision Making Within A Program, Conditions, Relational Operators, Logical Connectives, If Statement, If-Else Statement, Loops: While Loop, Do While, For Loop. Nested Loops, Infinite Loops, Switch Statement

**Functions:** Overview, defining a function, accessing a function, passing arguments to a function, specifying argument data types, function prototypes, recursion, modular programming and functions, standard library of c functions, prototype of a function: foo1lal parameter list, return type, function call, block structure, passing arguments to a function: call by reference, call by value

### **[3.1]Decision Making Within A Program:**

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 <b>if statement</b> consists of a boolean expression followed by one or more statements.
2	if...else statement  An <b>if statement</b> can be followed by an optional <b>else statement</b> , which executes when the Boolean expression is false.
3	nested if statements  You can use one <b>if</b> or <b>else if</b> statement inside another <b>if</b> or <b>else if</b> statement(s).
4	switch statement  A <b>switch</b> statement allows a variable to be tested for equality against a list of values.
5	nested switch statements  You can use one <b>switch</b> statement inside another <b>switch</b> statement(s).

### [3.2]Conditions, Relational Operators, Logical Connectives, If Statement, If-Else Statement, Loops: While Loop, Do While, For Loop. Nested Loops, Infinite Loops, Switch Statement.

#### Conditional operator:

##### 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.

### Relational Operators:-

The following table shows all the relational operators supported by C language. Assume variable **A** holds 10 and variable **B** holds 20 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.

### Example:-

Try the following example to understand all the relational operators available in C –

```
#include <stdio.h>

main() {

    int a = 21;
```

```
int b = 10;
int c ;

if( a == b ) {
    printf("Line 1 - a is equal to b\n" );
} else {
    printf("Line 1 - a is not equal to b\n" );
}

if ( a < b ) {
    printf("Line 2 - a is less than b\n" );
} else {
    printf("Line 2 - a is not less than b\n" );
}

if ( a > b ) {
    printf("Line 3 - a is greater than b\n" );
} else {
    printf("Line 3 - a is not greater than b\n" );
}

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

if ( a <= b ) {
    printf("Line 4 - a is either less than or equal to b\n" );
}

if ( b >= a ) {
    printf("Line 5 - b is either greater than or equal to b\n" );
}
}
```

When you compile and execute the above program, it produces the following result –

Line 1 - a is not equal to b  
Line 2 - a is not less than b  
Line 3 - a is greater than b  
Line 4 - a is either less than or equal to b  
Line 5 - b is either greater than or equal to b

**If Statement:-**

An **if** statement consists of a Boolean expression followed by one or more statements.

Syntax

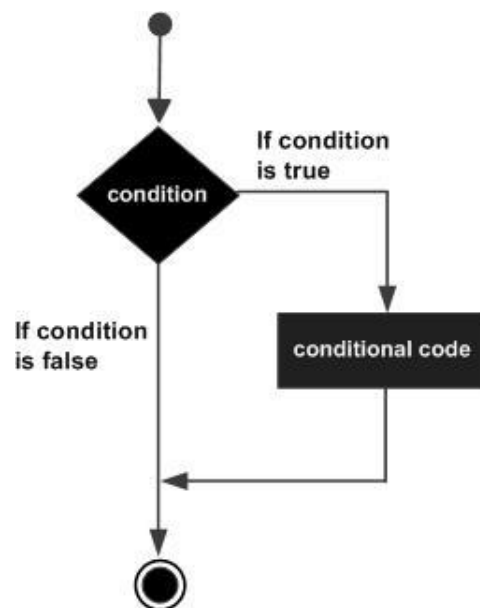
The syntax of an 'if' statement in C programming language is –

```
if(boolean_expression) {  
    /* statement(s) will execute if the boolean expression is true */  
}
```

If the Boolean expression evaluates to **true**, then the block of code inside the 'if' statement will be executed. If the Boolean expression evaluates to **false**, then the first set of code after the end of the 'if' statement (after the closing curly brace) will be executed.

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.

Flow Diagram:-



Example:-

```
#include <stdio.h>  
  
int main () {  
    /* local variable definition */  
    int a = 10;  
  
    /* check the boolean condition using if statement */  
    if( a < 20 ) {  
        /* if condition is true then print the following */  
        printf("a is less than 20\n" );  
    }  
  
    printf("value of a is : %d\n", a);  
    return 0;
```

```
}
```

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

a is less than 20;  
value of a is : 10

## If-Else Statement:

An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.

### Syntax:

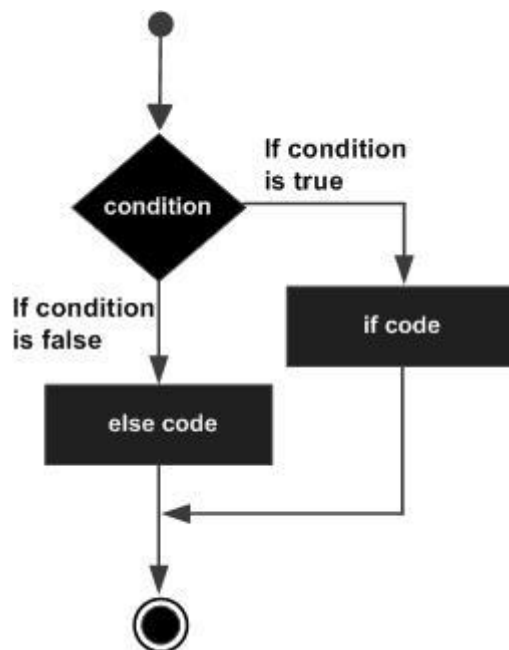
The syntax of an **if...else** statement in C programming language is –

```
if(boolean_expression) {  
    /* statement(s) will execute if the boolean expression is true */  
} else {  
    /* statement(s) will execute if the boolean expression is false */  
}
```

If the Boolean expression evaluates to **true**, then the **if block** will be executed, otherwise, the **else block** will be executed.

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.

### Flow Diagram:



Example



```

#include <stdio.h>

int main () {

    /* local variable definition */
    int a = 100;

    /* check the boolean condition */
    if( a < 20 ) {
        /* if condition is true then print the following */
        printf("a is less than 20\n" );
    } else {
        /* if condition is false then print the following */
        printf("a is not less than 20\n" );
    }

    printf("value of a is : %d\n", a);

    return 0;
}

```

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

```

a is not less than 20;
value of a is : 100

```

### If...else if...else Statement

An **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single if...else if statement.

When using if...else if..else statements, there are few points to keep in mind –

- An if can have zero or one else's and it must come after any if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

### Syntax

The syntax of an **if...else if...else** statement in C programming language is –

```

if(boolean_expression 1) {
    /* Executes when the boolean expression 1 is true */
} else if( boolean_expression 2) {
    /* Executes when the boolean expression 2 is true */
} else if( boolean_expression 3) {
    /* Executes when the boolean expression 3 is true */
} else {
    /* executes when the none of the above condition is true */
}

```

## Example

```
#include <stdio.h>

int main () {

    /* local variable definition */
    int a = 100;

    /* check the boolean condition */
    if( a == 10 ) {
        /* if condition is true then print the following */
        printf("Value of a is 10\n" );
    } else if( a == 20 ) {
        /* if else if condition is true */
        printf("Value of a is 20\n" );
    } else if( a == 30 ) {
        /* if else if condition is true */
        printf("Value of a is 30\n" );
    } else {
        /* if none of the conditions is true */
        printf("None of the values is matching\n" );
    }

    printf("Exact value of a is: %d\n", a );

    return 0;
}
```

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

```
None of the values is matching
Exact value of a is: 100
```

## LOOPS:

### 1)While Loop :

A **while** loop in C programming repeatedly executes a target statement as long as a given condition is true.

#### Syntax

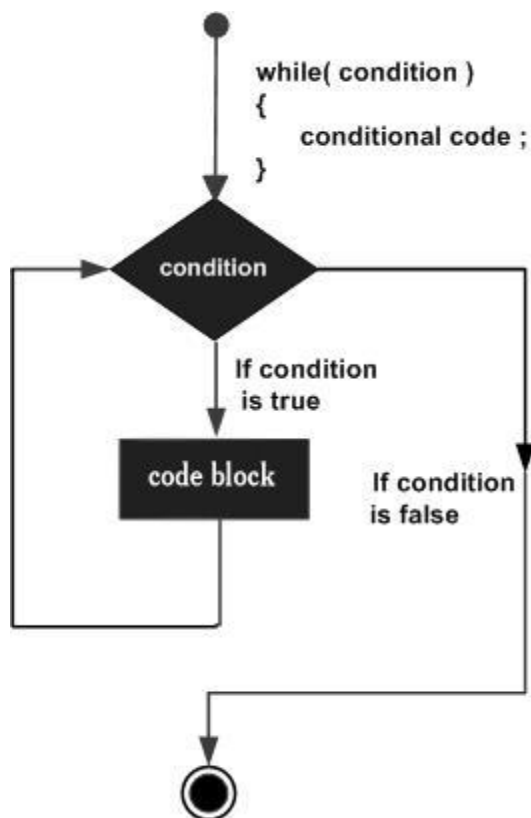
The syntax of a **while** loop in C programming language is –

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

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any nonzero value. The loop iterates while the condition is true.

When the condition becomes false, the program control passes to the line immediately following the loop.

### Flow Diagram:



Here, the key point to note is that a while loop might not execute at all. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

### Example

```
#include <stdio.h>

int main () {

    /* local variable definition */
    int a = 10;
```

```
/* while loop execution */  
while( a < 20 ) {  
    printf("value of a: %d\n", a);  
    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
```

## 2) Do While:

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop in C programming checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except the fact that it is guaranteed to execute at least one time.

### Syntax

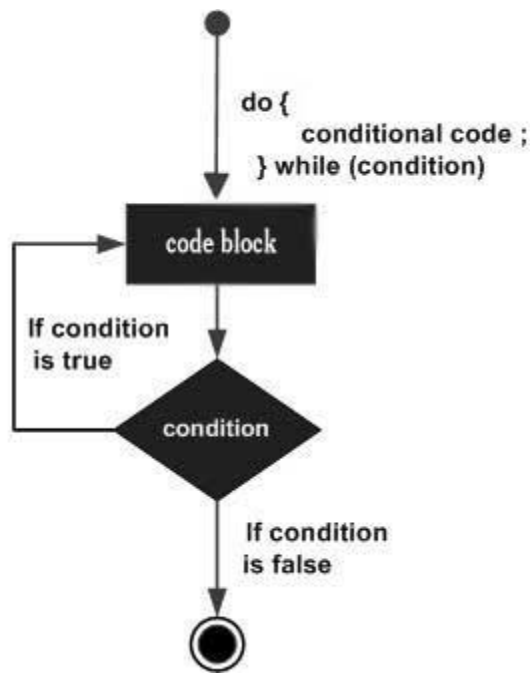
The syntax of a **do...while** loop in C programming language is –

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

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop executes once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop executes again. This process repeats until the given condition becomes false.

### Flow Diagram



Example

```
#include <stdio.h>

int main () {

    /* local variable definition */
    int a = 10;

    /* do loop execution */
    do {
        printf("value of a: %d\n", a);
        a = a + 1;
    }while( a < 20 );

    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
```

### 3) For Loop :

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

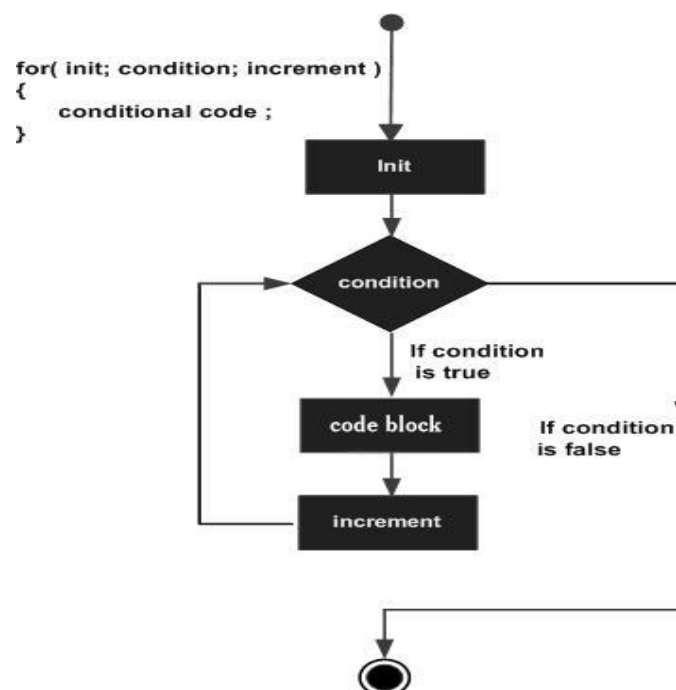
The syntax of a **for** loop in C programming language is –

```
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.

#### FLOW DIAGRAM:



## Example

```
#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
```

## 4) Nested Loops :

C supports nesting of loops in C. **Nesting of loops** is the feature in C that allows the looping of statements inside another loop. Let's observe an example of nesting loops in C.

Any number of loops can be defined inside another loop, i.e., there is no restriction for defining any number of loops. The nesting level can be defined at n times. You can define any type of loop inside another loop; for example, you can define '**while**' loop inside a '**for**' loop.

### Syntax of Nested loop

1. Outer\_loop
2. {
3.   Inner\_loop
4.   {

```
5.     // inner loop statements.
6. }
7.     // outer loop statements.
8. }
```

**Outer\_loop** and **Inner\_loop** are the valid loops that can be a 'for' loop, 'while' loop or 'do-while' loop.

### **Nested for loop**

The nested for loop means any type of loop which is defined inside the 'for' loop.

```
1. for (initialization; condition; update)
2. {
3.     for(initialization; condition; update)
4.     {
5.         // inner loop statements.
6.     }
7.     // outer loop statements.
8. }
```

### **Example of nested for loop**

```
1. #include <stdio.h>
2. int main()
3. {
4.     int n;// variable declaration
5.     printf("Enter the value of n :");
6.     // Displaying the n tables.
7.     for(int i=1;i<=n;i++) // outer loop
8.     {
9.         for(int j=1;j<=10;j++) // inner loop
10.        {
11.            printf("%d\t",(i*j)); // printing the value.
12.        }
13.        printf("\n");
14.    }
```

### **Explanation of the above code**

- First, the 'i' variable is initialized to 1 and then program control passes to the  $i \leq n$ .
- The program control checks whether the condition ' $i \leq n$ ' is true or not.
- If the condition is true, then the program control passes to the inner loop.
- The inner loop will get executed until the condition is true.



- After the execution of the inner loop, the control moves back to the update of the outer loop, i.e.,  $i++$ .
- After incrementing the value of the loop counter, the condition is checked again, i.e.,  $i \leq n$ .
- If the condition is true, then the inner loop will be executed again.
- This process will continue until the condition of the outer loop is true.

### Output:

```

input
Enter the value of n : 3
1      2      3      4      5      6      7      8      9      10
2      4      6      8      10     12     14     16     18     20
3      6      9      12     15     18     21     24     27     30

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

```

### Nested while loop

The nested while loop means any type of loop which is defined inside the 'while' loop.

1. **while**(condition)
2. {
3.   **while**(condition)
4.   {
5.     // inner loop statements.
6.   }
7.   // outer loop statements.
8. }

### Example of nested while loop

1. **#include** <stdio.h>
2. **int** main()
3. {
4.   **int** rows; // variable declaration
5.   **int** columns; // variable declaration
6.   **int** k=1; // variable initialization
7.   printf("Enter the number of rows :"); // input the number of rows.
8.   scanf("%d",&rows);
9.   printf("\nEnter the number of columns :"); // input the number of columns.
10.   scanf("%d",&columns);

```

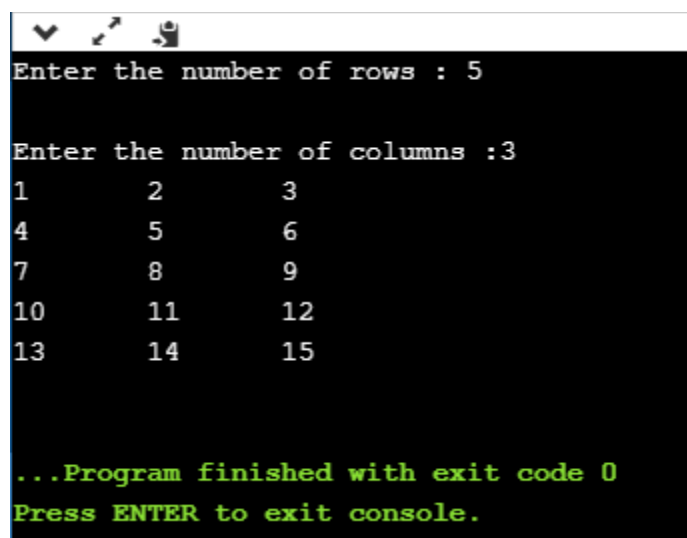
11.  int a[rows][columns]; //2d array declaration
12.  int i=1;
13.  while(i<=rows) // outer loop
14.  {
15.      int j=1;
16.      while(j<=columns) // inner loop
17.      {
18.          printf("%d\t",k); // printing the value of k.
19.          k++; // increment counter
20.          j++;
21.      }
22.      i++;
23.      printf("\n");
24.  }
25.}

```

### Explanation of the above code.

- We have created the 2d array, i.e., int a[rows][columns].
- The program initializes the 'i' variable by 1.
- Now, control moves to the while loop, and this loop checks whether the condition is true, then the program control moves to the inner loop.
- After the execution of the inner loop, the control moves to the update of the outer loop, i.e., i++.
- After incrementing the value of 'i', the condition (i<=rows) is checked.
- If the condition is true, the control then again moves to the inner loop.
- This process continues until the condition of the outer loop is true.

### Output:



```

Enter the number of rows : 5

Enter the number of columns :3
1      2      3
4      5      6
7      8      9
10     11     12
13     14     15

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

```

## Nested do..while loop

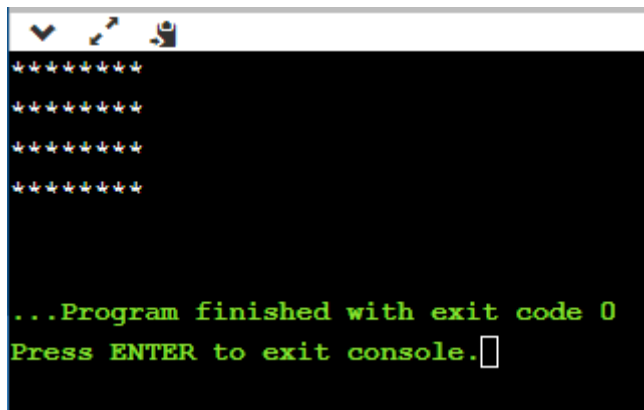
The nested do..while loop means any type of loop which is defined inside the 'do..while' loop.

```
1. do
2. {
3.     do
4.     {
5.         // inner loop statements.
6.     }while(condition);
7. // outer loop statements.
8. }while(condition);
```

### Example of nested do..while loop.

```
1. #include <stdio.h>
2. int main()
3. {
4.     /*printing the pattern
5.     *****
6.     *****
7.     *****
8.     ***** */
9.     int i=1;
10.    do        // outer loop
11.    {
12.        int j=1;
13.        do    // inner loop
14.        {
15.            printf("*");
16.            j++;
17.        }while(j<=8);
18.        printf("\n");
19.        i++;
20.    }while(i<=4);
21. }
```

### Output:

A screenshot of a console window with a black background and green text. At the top, there are four rows of ten star characters (☆☆☆☆☆☆☆☆) each. Below these, the text reads: "...Program finished with exit code 0" followed by "Press ENTER to exit console." with a cursor icon at the end.

### Explanation of the above code.

- First, we initialize the outer loop counter variable, i.e., 'i' by 1.
- As we know that the do..while loop executes once without checking the condition, so the inner loop is executed without checking the condition in the outer loop.
- After the execution of the inner loop, the control moves to the update of the i++.
- When the loop counter value is incremented, the condition is checked. If the condition in the outer loop is true, then the inner loop is executed.
- This process will continue until the condition in the outer loop is true.

## 5) Infinite Loops :

What is infinite loop?

An infinite loop is a looping construct that does not terminate the loop and executes the loop forever. It is also called an **indefinite** loop or an **endless** loop. It either produces a continuous output or no output.

When to use an infinite loop

An infinite loop is useful for those applications that accept the user input and generate the output continuously until the user exits from the application manually. In the following situations, this type of loop can be used:

- All the operating systems run in an infinite loop as it does not exist after performing some task. It comes out of an infinite loop only when the user manually shuts down the system.

- All the servers run in an infinite loop as the server responds to all the client requests. It comes out of an indefinite loop only when the administrator shuts down the server manually.
- All the games also run in an infinite loop. The game will accept the user requests until the user exits from the game.

We can create an infinite loop through various loop structures. The following are the loop structures through which we will define the infinite loop

- for loop
- while loop
- do-while loop
- go to statement
- C macros

### For loop

Let's see the **infinite 'for'** loop. The following is the definition for the **infinite** for loop:

```
1. for(;;)
2. {
3.     // body of the for loop.
4. }
```

As we know that all the parts of the **'for' loop** are optional, and in the above for loop, we have not mentioned any condition; so, this loop will execute infinite times.

**Let's understand through an example.**

```
1. #include <stdio.h>
2. int main()
3. {
4.     for(;;)
5.     {
6.         printf("Hello javatpoint");
7.     }
8.     return 0;
9. }
```

In the above code, we run the 'for' loop infinite times, so **"Hello javatpoint"** will be displayed infinitely.

**Output:**



A screenshot of a terminal window with a dark background. The title bar at the top shows standard window controls and the word 'input'. The terminal displays a continuous stream of the text 'javatpointHello' repeated many times, with each repetition on a new line. The text is in a light-colored, monospaced font.

## 6)Switch Statement:

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each **switch case**.

Syntax

The syntax for a **switch** statement in C programming language is as follows –

```
switch(expression) {

    case constant-expression :
        statement(s);
        break; /* optional */

    case constant-expression :
        statement(s);
        break; /* optional */

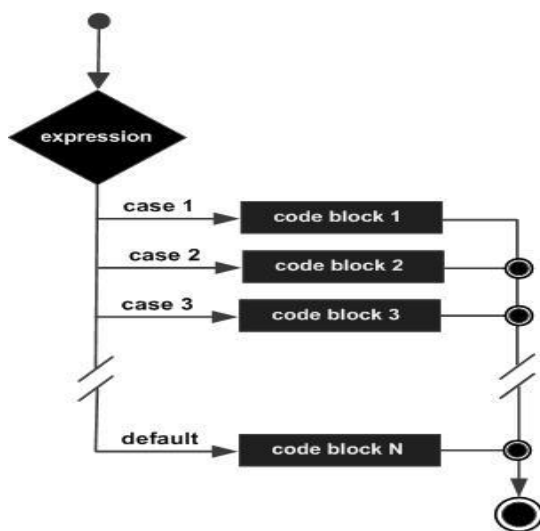
    /* you can have any number of case statements */
    default: /* Optional */
        statement(s);
}
```

The following rules apply to a **switch** statement –

- The **expression** used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.

- The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.
- When a **break** statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a **break**. If no **break** appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No **break** is needed in the default case.

### Flow Diagram



### Example

```

#include <stdio.h>

int main () {

    /* local variable definition */
    char grade = 'B';

    switch(grade) {
        case 'A' :
            printf("Excellent!\n" );
            break;
        case 'B' :
        case 'C' :
            printf("Well done\n" );
    }
}

```

```
        break;
    case 'D' :
        printf("You passed\n");
        break;
    case 'F' :
        printf("Better try again\n");
        break;
    default :
        printf("Invalid grade\n");
}

printf("Your grade is %c\n", grade);

return 0;
}
```

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

Well done

Your grade is B



## UNIT 3: CHAPTER-2

### Functions

Overview, defining a function, accessing a function, passing arguments to a function, specifying argument data types, function prototypes, recursion, modular programming and functions, standard library of c functions, prototype of a function: foo1lal parameter list, return type, function call, block structure, passing arguments to a function: call by reference, call by value

#### Overview:

A 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 is such that 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, **strcat()** to concatenate two strings, **memcpy()** to copy one memory location to another location, and many more functions.

A function can also be referred as 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 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**.
- **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

Given below is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum value 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(), the function declaration is as follows –

int max(int num1, int num2);

Parameter names are not important in function declaration only their type is required, so the following is also a 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, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

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

```
#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;
}
```

We have kept max() along with main() and compiled the source code. While running the 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.

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 in which arguments can be passed to a function –

Sr.No.	Call Type & Description
1	<u>Call by value</u>  This method 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.
2	<u>Call by reference</u>  This method 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 argument.

By default, C uses **call by value** to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

### function prototypes:

A function prototype is one of the most important features of C programming which was originated from C++. A function prototype is a declaration in the code that instructs the compiler about the data type of the function, arguments and parameter list. As we all know that a block of code which performs a specific task is called as a function. In the same way, a function prototype is a function which specifies return type, function name and its parameter to the compiler so that it can match with the given function calls when required.

### Syntax:

```
return type function name( datatype paramter1 , datatype paramter2 , datatype  
paramter3..);
```

### **Example:**

### **Code:**

```
int addition(int a, int b);
```

In the above example addition is the name of the function of integer data type is the return type and a and b are the argument of two arguments of type int passed to the function. Note that we can pass as many arguments we want in our function based on the requirement. Also in the same program we can define as many prototype we want but they should differ in either name or argument list. All you have to do is define a prototype in the code and then call it anytime by using the function name.

### **Examples of Function Prototype in C**

Given below are the examples mentioned:

### **Code:**

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

```
int Num_subtraction( inti , int j ); // prototype for the function
```

```
int main()
```

```
{
```

```
int num1 , num2 , output ;
```

```

printf( " Please enters the 2 numbers you want to subtract : " );

scanf( "%d %d" , &num1 , &num2 );

output = Num_subtraction( num1 , num2 );

printf( " The subtraction of the given numbers is = %d " , output );

return 0 ;

}

intNum_subtraction( inti , int j )// function definition

{

intresults ;

results = i - j ;

return results ;

}

```

### Output:

```

Please enters the 2 numbers you want to subtract : 90
45
The subtraction of the given numbers is = 45

```

As you can see in the above code, initially we are declaring the function prototype for the subtraction of two numbers with name “ Num\_subtraction ” of integer return type with two integer arguments named as i and j into the function. In the main class, we defined three integers num1, num2, and output. After that, we are taking input from the

users then storing the subtraction results of the two given numbers in output. To call the function Num\_subtraction function is used again. At last in the function definition you can see we are giving the logic to perform subtraction and store it in results.

## 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 () {  
    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

The following example calculates the factorial of a given number using a recursive function –

```
#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;  
}
```

```
}
```

When the above code is compiled and executed, it produces the following result –  
Factorial of 12 is 479001600

### Fibonacci Series

The following example generates the Fibonacci series for a given number using a recursive function –

```
#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", 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
```



## Modular Programming And Functions,

Modular programming is the process of subdividing a computer program into separate sub-programs. A module is a separate software component. It can often be used in a variety of applications and functions with other components of the system.

- Some programs might have thousands or millions of lines and to manage such programs it becomes quite difficult as there might be too many of syntax errors or logical errors present in the program, so to manage such type of programs concept of **modular programming** approached.
- Each sub-module contains something necessary to execute only one aspect of the desired functionality.
- Modular programming emphasis on breaking of large programs into small problems to increase the maintainability, readability of the code and to make the program handy to make any changes in future or to correct the errors.

### **Points which should be taken care of prior to modular program development:**

1. Limitations of each and every module should be decided.
2. In which way a program is to be partitioned into different modules.
3. Communication among different modules of the code for proper execution of the entire program.

### **Advantages of Using Modular Programming Approach –**

1. **Ease of Use** :This approach allows simplicity, as rather than focusing on the entire thousands and millions of lines code in one go we can access it in the form of modules. This allows ease in debugging the code and prone to less error.
2. **Reusability** :It allows the user to reuse the functionality with a different interface without typing the whole program again.
3. **Ease of Maintenance** : It helps in less collision at the time of working on modules, helping a team to work with proper collaboration while working on a large application.

### **Example of Modular Programming in C**

C is called a structured programming language because to solve a large problem, C programming language divides the problem into smaller modules called functions or procedures each of which handles a particular responsibility. The program which solves the entire problem is a collection of such functions.

Module is basically a set of interrelated files that share their implementation details but hide it from the outside world. How can we implement modular programming in c?

Each function defined in C by default is globally accessible. This can be done by including the header file in which implementation of the function is defined.

Suppose, we want to declare a **Stack** data type and at the same time want to hide the implementation, including its data structure, from users. We can do this by first defining a public file called **stack.h** which contains generic data Stack data type and the functions which are supported by the stack data type.

In the **header file** we must include only the definitions of constants, structures, variables and functions with the name of the module, that makes easy to identify source

of definition in a larger program with many modules.

Keywords **extern** and **static** help in the implementation of modularity in C.

stack.h:

```
extern stack_var1;
extern int stack_do_something(void);
```

Now we can create a file named **stack.c** that contains implementation of stack data type:

```
#include
int stack_var1;
static int stack_var2;

int stack_do_something(void)
{
    stack_var1 = 2;
    stack_var2 = 5;
}
```

The main file which may includes module stack

```
#include
int main(int argc, char*argv[]){
while(1){
    stack_do_something();
    }
}
```

## Standard Library Of C Functions:

### 1. Standard Library Functions in C

Standard Library Functions are basically the inbuilt functions in the C compiler that makes things easy for the programmer.

As we have already discussed, every C program has at least one function, that is, the **main() function**. The main() function is also a standard library function in C since it is inbuilt and conveys a specific meaning to the C compiler.

### 2. Significance of Standard Library Functions in C

## 2.1 Usability

Standard library functions allow the programmer to use the pre-existing codes available in the C compiler without the need for the user to define his own code by deriving the logic to perform certain basic functions.

## 2.2 Flexibility

A wide variety of programs can be made by the programmer by making slight modifications while using the standard library functions in C.

## 2.3 User-friendly syntax

We have already discussed in Function in C tutorial that how easy it is to grasp and use the syntax of functions.

## 2.4 Optimization and Reliability

All the standard library functions in C have been tested multiple times in order to generate the optimal output with maximum efficiency making it reliable to use.

## 2.5 Time-saving

Instead of writing numerous lines of codes, these functions help the programmer to save time by simply using the pre-existing functions.

## 2.6 Portability

Standard library functions are available in the C compiler irrespective of the device you are working on. These functions connote the same meaning and hence serve the same purpose regardless of the operating system or programming environment.

## 3. Header Files in C:

In order to access the standard library functions in C, certain header files need to be included before writing the body of the program.

Here is a tabular representation of a list of header files associated with some of the standard library functions in C:

HEADER FILE	MEANING	ELUCIDATION
<b>&lt;stdio.h&gt;</b>	Standard input-output header	Used to perform input and output operations like <a href="#">scanf()</a> and <a href="#">printf()</a> .
<b>&lt;string.h&gt;</b>	String header	Used to perform string manipulation operations like <a href="#">strlen</a> and <a href="#">strcpy</a> .

<b>&lt;conio.h&gt;</b>	Console input-output header	Used to perform console input and console output operations like clrscr() to clear the screen and getch() to get the character from the keyboard.
<b>&lt;stdlib.h&gt;</b>	Standard library header	Used to perform standard utility functions like dynamic memory allocation using functions such as malloc() and calloc().
<b>&lt;math.h&gt;</b>	Math header	Used to perform mathematical operations like sqrt() and pow() to obtain the square root and the power of a number respectively.
<b>&lt;ctype.h&gt;</b>	Character type header	Used to perform character type functions like isalpha() and isdigit() to find whether the given character is an alphabet or a digit respectively.
<b>&lt;time.h&gt;</b>	Time header	Used to perform functions related to date and time like setdate() and getdate() to modify the system date and get the CPU time respectively.
<b>&lt;assert.h&gt;</b>	Assertion header	Used in program assertion functions like assert() to get an integer data type as a parameter which prints stderr only if the parameter passed is 0.
<b>&lt;locale.h&gt;</b>	Localization header	Used to perform localization functions like setlocale() and localeconv() to set locale and get locale conventions respectively.
<b>&lt;signal.h&gt;</b>	Signal header	Used to perform signal handling functions like signal() and raise() to install signal handler and to raise the signal in the program respectively.
<b>&lt;setjmp.h&gt;</b>	Jump header	Used to perform jump functions.
<b>&lt;stdarg.h&gt;</b>	Standard argument header	Used to perform standard argument functions like va_start and va_arg() to indicate the start of the variable-length argument list and to fetch the arguments from the variable-length argument list in the program respectively.
<b>&lt;errno.h&gt;</b>	Error handling header	Used to perform error handling operations like errno() to indicate errors in the program by initially assigning the value of this function to 0 and then later changing it to indicate errors.

## Prototype Of A Function:

### Block Structure:

In computer programming, a block or code block is a lexical structure of source code which is grouped together. Blocks consist of one or more declarations and statements. A programming language that permits the creation of blocks, including blocks nested within other blocks, is called a block-structured programming language. Blocks are fundamental to structured programming, where control structures are formed from blocks.

The function of blocks in programming is to enable groups of statements to be treated as if they were one statement, and to narrow the lexical scope of objects such as variables, procedures and functions declared in a block so that they do not conflict with those having the same name used elsewhere. In a block-structured programming language, the objects named in outer blocks are visible inside inner blocks, unless they are masked by an object declared with the same name.

## Passing Arguments To A Function:

### Call By Reference, Call By Value:

#### *Call By Value Method*

In Call By Value method, the arguments are passed to a function using the **value of the variable**. There is a second copy of the variable is formed while passing values from one function to another.

Therefore, the original values or the **original variables do not change**. The call by value and call by reference approach are mostly used while passing values to functions in C programming.

The original variables are called **Actual Parameters** whereas the new values that are passed to the function are called as **Formal Parameters**.

In case of call by value approach, the copy of the values are passed to the function. The actual parameters and formal parameters are thereby created in different memory locations. Therefore, the original value is not modified in call by value method in C programming.

The values passed to the **User Defined Function** are created on **Stack** memory location. Therefore, the changes made within the user defined function is not visible in the **main()** method or any other method for that matter.

The disadvantage with pass by value method is that there are two copies created for the same variable which is not **memory efficient**. However, the advantage of call by value approach is that this method doesn't change the original variables thereby preserving

data. The following program will demonstrate how to write a C code to swap two variables using call by value method.

In the following C program to swap variables with Call By Value approach, the variables **var1** and **var2** are passed to the **swap()** method by using their values and not the addresses.

When the user defined function **swap()** receives the values of var1 and var2, two local copies are created for that variables in the function. These local values **val1** and **val2** are manipulated by the program itself and there is no effect on the actual parameters var1 and var2 that were passed from the main() function.

**Example:** [C Program To Swap Two Variables using Call By Value Approach](#)

### **C Program To Swap Variables using Call by Value Method:**

```
#include<stdio.h>

int swap(int a, int b);

int main()
{
    int num1, num2;
    printf("\nEnter The First Number:\t");
    scanf("%d", &num1);
    printf("\nEnter The Second Number:\t");
    scanf("%d", &num2);
    swap(num1, num2);
    printf("\nOld Values\n");
    printf("\nFirst Number = %d\nSecond Number = %d\n", num1, num2);
    printf("\n");
    return 0;
}

int swap(int a, int b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
```

```
printf("\nNew Values\n");  
printf("\nFirst Number = %d\nSecond Number = %d\n", a, b);  
}
```

### Call By Reference Method:

In Call By Reference method, the arguments are passed to another function using the **address of the variables**. This address is also known as **Reference Pointers**. There is no second copy of the variable while passing values from one function to another.

Since, the original variables' address is passed, the modifications done to the variable in the User Defined will **change the original variable** as well. Therefore, the Original Values will change.

The original variables are called **Actual Parameters** whereas the new values that are created within the user defined function are called **Formal Parameters**. In case of call by reference approach, the address of the original values are passed to the function. The actual parameters and formal parameters are thereby created in the **same memory location**. All the modification is done on actual parameters. Therefore, the original value is modified in Pass By Reference method in C programming.

The changes made within the user defined function are visible in the **main()** method and any other method for that matter. We have demonstrated below a C code for call by reference in C using Pointers.

The Advantage of using pass by reference approach is that it does not create duplicate data for holding only one value. Therefore, this helps to **save memory space**. The following program will demonstrate how to write a C program to swap two variables using pass by reference method.

In the C program to swap variables with passing by reference approach, the variables **var1** and **var2** are passed to the **swap()** method by using their addresses and not the values.

When the User Defined Function **swap()** receives the addresses of var1 and var2, the pointers **\*ptr1** and **\*ptr2** holds the values of the var1 and var2 since the addresses are passed to them. Therefore, no local copies are created. The original variables var1 and var2 will be directly modified by ptr1 and ptr2.

**Example: Swapping Two Numbers using Call By Reference approach in C Programming**

### C Program To Swap Variables using Call By Reference Method

```
#include<stdio.h>  
  
int swap(int *ptr1, int *pt2);
```

```
int main()
{
    int var1, var2;
    printf("Enter The First Number:\t");
    scanf("%d", &var1);
    printf("Enter The Second Number:\t");
    scanf("%d", &var2);
    swap(&var1, &var2);

    printf("The Swapped Values are:\n");
    printf("First Number = %d\nSecond Number = %d\n", var1, var2);
```

```
    return 0;
}

int swap(int *ptr1, int *ptr2)
{
    int temp;
    temp = *ptr1;
    *ptr1 = *ptr2;
    *ptr2 = temp;
}
```



## **UNIT-4:CHAPTER-1**

**Program structure:**Storage classes, automatic variables, external variables, static variables, multifile programs, more library functions,

**Preprocessor:** Features, #define and #include, Directives and Macros

**Arrays:**Definition, processing, passing arrays to functions, multidimensional arrays, arrays and strings

### **Program Structure**

This chapter is concerned with the structure of programs consisting of more than one function. We will first consider the distinction between “local” variables that are recognized only within a single function, and “global” variables that are recognized in two or more functions. We will see how global variables are defined and utilized in this chapter. We will also consider the issue of static vs. dynamic retention of information by a local variable. That is, a local variable normally does not retain its value once control has been transferred out of its defining function. In some circumstances, however, it may be desirable to have certain local variables retain their values, so that the function can be re-entered at a later time and the computation resumed. And finally, it may be desirable to develop a large, multifunction program in terms of several independent files, with a small number of functions (perhaps only one) defined within each file. In such programs the individual functions can be defined and accessed locally within a single file, or globally within multiple files. This is similar to the definition and use of local vs. global variables in a multifunction, single-file program.

### **Storage classes:**

Storage Classes are used to describe the features of a variable/function. These features basically include the scope, visibility and life-time which help us to trace the existence of a particular variable during the runtime of a program.

**C language uses 4 storage classes, namely:**

#### **1. automatic variables:**

**auto:** This is the default storage class for all the variables declared inside a function or a block. Hence, the keyword auto is rarely used while writing programs in C language. Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope). Of course, these can be accessed within nested blocks within the parent block/function in which the auto variable was declared. However, they can be accessed outside their scope as well using the concept of pointers given here by pointing to the very exact memory location where the variables resides. They are assigned a garbage value by default whenever they are declared.

## 2.external variables:

**extern**: Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well. So an extern variable is nothing but 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. Also, a normal global variable can be made extern as well by placing the 'extern' keyword before its declaration/definition in any function/block. This basically signifies that we are not initializing a new variable but instead we are using/accessing the global variable only. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program. For more information on how extern variables work, have a look at this [link](#).

## 3.static variables:

**static**: This storage class is used to declare static variables which are popularly used while writing programs in C language. Static variables have a property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope. So we can say that they are initialized only once and exist till the termination of the program. Thus, 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, they are assigned the value 0 by the compiler.

## 4. register variable:

**register**: This storage class declares register variables which have the same functionality as that of the auto variables. 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 runtime of the program. If a free register is not available, these are then stored in the memory only. Usually 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 and interesting 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:**

<b>storage_class var_data_type var_name;</b>
--

## Preprocessor: Features, #define and #include, Directives and Macros

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. We'll refer to the C Preprocessor as CPP.

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 following section lists down all the important preprocessor directives –

Sr.No.	Directive & Description
1	<b>#define</b> Substitutes a preprocessor macro.
2	<b>#include</b> Inserts a particular header from another file.
3	<b>#undef</b> Undefines a preprocessor macro.
4	<b>#ifdef</b> Returns true if this macro is defined.
5	<b>#ifndef</b> Returns true if this macro is not defined.
6	<b>#if</b> Tests if a compile time condition is true.
7	<b>#else</b> The alternative for #if.
8	<b>#elif</b>

	<code>#else</code> and <code>#if</code> in one statement.
9	<b><code>#endif</code></b> Ends preprocessor conditional.
10	<b><code>#error</code></b> Prints error message on stderr.
11	<b><code>#pragma</code></b> Issues special commands to the compiler, using a standardized method.

## Preprocessors Examples

Analyze the following examples to understand various directives.

```
#define MAX_ARRAY_LENGTH 20
```

This directive tells the CPP 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 the CPP to get `stdio.h` from **System Libraries** and add the text to the current source file. The next line tells CPP to get **`myheader.h`** from the local directory and add the content to the current source file.

```
#undef FILE_SIZE
#define FILE_SIZE 42
```

It tells the CPP to undefine existing `FILE_SIZE` and define it as `42`.

```
#ifndef MESSAGE
#define MESSAGE "You wish!"
#endif
```

It tells the CPP to define `MESSAGE` only if `MESSAGE` isn't already defined.

```
#ifdef DEBUG
/* Your debugging statements here */
#endif
```

It tells the CPP 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

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	<b>__DATE__</b> The current date as a character literal in "MMM DD YYYY" format.
2	<b>__TIME__</b> The current time as a character literal in "HH:MM:SS" format.
3	<b>__FILE__</b> This contains the current filename as a string literal.
4	<b>__LINE__</b> This contains the current line number as a decimal constant.
5	<b>__STDC__</b> Defined as 1 when the compiler complies with the ANSI standard.

Let's try the following example –

```
#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 :Jun 2 2012

Time :03:36:24

Line :8

## 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 –

```
#include <stdio.h>  
  
#define message_for(a, b) \  
    printf("#a " and " #b ": We love you!\n")  
  
int main(void) {  
    message_for(Carole, Debra);  
    return 0;  
}
```

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

Carole and Debra: 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 –

```
#include <stdio.h>  
  
#define tokenpaster(n) printf ("token" #n " = %d", token##n)  
  
int main(void) {  
    int token34 = 40;  
    tokenpaster(34);  
}
```

```
return 0;  
}
```

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

token34 = 40

It happened so because this example results in the following actual output from the preprocessor –

```
printf ("token34 = %d", token34);
```

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 –

```
#include <stdio.h>  
  
#if !defined (MESSAGE)  
    #define MESSAGE "You wish!"  
#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 wish!

### Parameterized Macros

One of the powerful functions of the CPP 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 x) {  
    return x * x;  
}
```

We can rewrite above the code using a macro as follows –

```
#define square(x) ((x) * (x))
```

Macros with arguments must be defined using the **#define** directive before they can be used. 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 –

```
#include <stdio.h>

#define MAX(x,y) ((x) > (y) ? (x) : (y))

int main(void) {
    printf("Max between 20 and 10 is %d\n", MAX(10, 20));
    return 0;
}
```

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

Max between 20 and 10 is 20

## **Arrays: Definition, processing, passing arrays to functions, multidimensional arrays, arrays and strings:-**

### **C Array**

An array is defined as the collection of similar type of data items stored at contiguous memory locations. Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc. It also has the capability to store the collection of derived data types, such as pointers, structure, etc. The array is the simplest data structure where each data element can be randomly accessed by using its index number.

C array is beneficial if you have to store similar elements. For example, if we want to store the marks of a student in 6 subjects, then we don't need to define different variables for the marks in the different subject. Instead of that, we can define an array which can store the marks in each subject at the contiguous memory locations.

By using the array, we can access the elements easily. Only a few lines of code are required to access the elements of the array.

### **Properties of Array**

The array contains the following properties.

- Each element of an array is of same data type and carries the same size, i.e., int = 4 bytes.
- Elements of the array are stored at contiguous memory locations where the first element is stored at the smallest memory location.



- Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

### Advantage of C Array

- 1) Code Optimization:** Less code to access the data.
- 2) Ease of traversing:** By using the for loop, we can retrieve the elements of an array easily.
- 3) Ease of sorting:** To sort the elements of the array, we need a few lines of code only.
- 4) Random Access:** We can access any element randomly using the array.

### Disadvantage of C Array

- 1) Fixed Size:** Whatever size, we define at the time of declaration of the array, we can't exceed the limit. So, it doesn't grow the size dynamically like LinkedList which we will learn later.

### Declaration of C Array

We can declare an array in the C language in the following way.

1. `data_type array_name[array_size];`

Now, let us see the example to declare the array.

1. `int marks[5];`

Here, `int` is the *data\_type*, `marks` are the *array\_name*, and `5` is the *array\_size*.

### Initialization of C Array

The simplest way to initialize an array is by using the index of each element. We can initialize each element of the array by using the index. Consider the following example.

1. `marks[0]=80; //initialization of array`
2. `marks[1]=60;`
3. `marks[2]=70;`
4. `marks[3]=85;`
5. `marks[4]=75;`

80	60	70	85	75
----	----	----	----	----

marks[0]   marks[1]   marks[2]   marks[3]   marks[4]

## Initialization of Array

### C array example

```
1. #include<stdio.h>
2. int main(){
3. int i=0;
4. int marks[5]; //declaration of array
5. marks[0]=80; //initialization of array
6. marks[1]=60;
7. marks[2]=70;
8. marks[3]=85;
9. marks[4]=75;
10. //traversal of array
11. for(i=0;i<5;i++){
12. printf("%d \n",marks[i]);
13. } //end of for loop
14. return 0;
15. }
```

### Output

```
80
60
70
85
75
```

### C Array: Declaration with Initialization

We can initialize the c array at the time of declaration. Let's see the code.

```
1. int marks[5]={20,30,40,50,60};
```

In such case, there is **no requirement to define the size**. So it may also be written as the following code.

```
1. int marks[]={20,30,40,50,60};
```

Let's see the C program to declare and initialize the array in C.

### One dimensional array in C:

```
1. #include<stdio.h>
2. int main(){
3. int i=0;
4. int marks[5]={20,30,40,50,60}; //declaration and initialization of array
5. //traversal of array
6. for(i=0;i<5;i++){
7. printf("%d \n",marks[i]);
8. }
9. return 0;
10.}
```

### Output

```
20
30
40
50
60
```

### Two Dimensional Array in C

The two-dimensional array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns. However, 2D arrays are created to implement a relational database lookalike data structure. It provides ease of holding the bulk of data at once which can be passed to any number of functions wherever required.

### Declaration of two dimensional Array in C

The syntax to declare the 2D array is given below.

```
1. data_type array_name[rows][columns];
```

Consider the following example.

```
1. int twodimen[4][3];
```

Here, 4 is the number of rows, and 3 is the number of columns.

### Initialization of 2D Array in C

In the 1D array, we don't need to specify the size of the array if the declaration and initialization are being done simultaneously. However, this will not work with 2D arrays. We will have to define at least the second dimension of the array. The two-dimensional array can be declared and defined in the following way.

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

Two-dimensional array example in C

```
1. #include<stdio.h>
2. int main(){
3.     int i=0,j=0;
4.     int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
5.     //traversing 2D array
6.     for(i=0;i<4;i++){
7.         for(j=0;j<3;j++){
8.             printf("arr[%d] [%d] = %d \n",i,j,arr[i][j]);
9.         }//end of j
10.    }//end of i
11.    return 0;
12.}
```

### Output

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

## UNIT 5 :CHAPTER -1

Pointers: Fundamentals, declarations, Pointers Address Operators, Pointer Type Declaration, Pointer Assignment, Pointer Initialization, Pointer Arithmetic, Functions and Pointers, Arrays And Pointers, Pointer Arrays, passing functions to other functions

### What are Pointers?

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is –

type \*var-name;

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk \* used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations –

```
int *ip; /* pointer to an integer */
double *dp; /* pointer to a double */
float *fp; /* pointer to a float */
char *ch /* pointer to a character */
```

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.

### How to Use Pointers?

There are a few important operations, which we will do with the help of pointers very frequently. **(a)** We define a pointer variable, **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator \* that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations –

```
#include <stdio.h>

int main () {

    int var = 20; /* actual variable declaration */
    int *ip;      /* pointer variable declaration */
```

```

ip = &var; /* store address of var in pointer variable*/

printf ("Address of var variable: %x\n", &var );

/* address stored in pointer variable */
printf ("Address stored in ip variable: %x\n", ip );

/* access the value using the pointer */
printf ("Value of *ip variable: %d\n", *ip );

return 0;
}

```

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

```

Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20

```

## NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program –

```

#include <stdio.h>

int main () {

    int *ptr = NULL;

    printf ("The value of ptr is : %x\n", ptr );

    return 0;
}

```

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

```

The value of ptr is 0

```

In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer, you can use an 'if' statement as follows –

```
if(ptr) /* succeeds if p is not null */  
if(!ptr) /* succeeds if p is null */
```

## Pointers in Detail

Pointers have many but easy concepts and they are very important to C programming. The following important pointer concepts should be clear to any C programmer –

Sr.No.	Concept & Description
1	<u>Pointer arithmetic</u>  There are four arithmetic operators that can be used in pointers: ++, --, +, -
2	<u>Array of pointers</u>  You can define arrays to hold a number of pointers.
3	<u>Pointer to pointer</u>  C allows you to have pointer on a pointer and so on.
4	<u>Passing pointers to functions in C</u>  Passing an argument by reference or by address enable the passed argument to be changed in the calling function by the called function.
5	<u>Return pointer from functions in C</u>  C allows a function to return a pointer to the local variable, static variable, and dynamically allocated memory as well.

### Pointer arithmetic:

A pointer in c is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -

To understand pointer arithmetic, let us consider that **ptr** is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer –

```
ptr++
```

After the above operation, the **ptr** will point to the location 1004 because each time ptr is incremented, it will point to the next integer location which is 4 bytes next to the

current location. This operation will move the pointer to the next memory location without impacting the actual value at the memory location. If **ptr** points to a character whose address is 1000, then the above operation will point to the location 1001 because the next character will be available at 1001.

### 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;
}
```

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

```
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;
}
```

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

```
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 Array:**

It is most likely that you would not understand this section until you are through with the chapter 'Pointers'.

Assuming you have some understanding of pointers in C, let us start: An array name is a constant pointer to the first element of the array. Therefore, in the declaration –

```
double balance[50];
```

**balance** is a pointer to &balance[0], which is the address of the first element of the array balance. Thus, the following program fragment assigns **p** as the address of the first element of **balance** –

```
double *p;  
double balance[10];
```

```
p = balance;
```

It is legal to use array names as constant pointers, and vice versa. Therefore, \*(balance + 4) is a legitimate way of accessing the data at balance[4].

Once you store the address of the first element in 'p', you can access the array elements using \*p, \*(p+1), \*(p+2) and so on. Given below is the example to show all the concepts discussed above –

```
#include <stdio.h>  
  
int main () {  
  
    /* an array with 5 elements */  
    double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};  
    double *p;  
    int i;  
  
    p = balance;  
  
    /* output each array element's value */  
    printf ( "Array values using pointer\n" );  
  
    for ( i = 0; i < 5; i++ ) {  
        printf ( "(p + %d) : %f\n", i, * (p + i) );  
    }  
  
    printf ( "Array values using balance as address\n" );  
  
    for ( i = 0; i < 5; i++ ) {  
        printf ( "(balance + %d) : %f\n", i, * (balance + i) );  
    }  
  
    return 0;  
}
```

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

Array values using pointer

\*(p + 0) : 1000.000000

\*(p + 1) : 2.000000

\*(p + 2) : 3.400000

\*(p + 3) : 17.000000

\*(p + 4) : 50.000000

Array values using balance as address

\*(balance + 0) : 1000.000000

\*(balance + 1) : 2.000000

\*(balance + 2) : 3.400000

\*(balance + 3) : 17.000000

\*(balance + 4) : 50.000000

In the above example, p is a pointer to double, which means it can store the address of a variable of double type. Once we have the address in p, **\*p** will give us the value available at the address stored in p, as we have shown in the above example

## UNIT-5:CHAPTER-2

Structures and Unions: Structure Variables, Initialization, Structure Assignment, Nested Structure, Structures and Functions, Structures and Arrays: Arrays of Structures, Structures Containing Arrays, Unions, Structures and pointers.

Arrays allow to define type of variables that can hold several data items of the same kind. Similarly **structure** is another user defined data type available in C that allows to combine data items of different kinds.

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book –

- Title
- Author
- Subject
- Book ID

### Defining a Structure

To define a structure, you must use the **struct** statement. The struct statement defines a new data type, with more than one member. The format of the struct statement is as follows –

```
struct [structure tag] {  
    member definition;  
    member definition;  
    ...  
    member definition;  
} [one or more structure variables];
```

The **structure tag** is optional and each member definition is a normal variable definition, such as `int i;` or `float f;` or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure –

```
struct Books {  
    char title[50];  
    char author[50];  
    char subject[100];  
    int book_id;  
} book;
```

## Accessing Structure Members:

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use the keyword **struct** to define variables of structure type. The following example shows how to use a structure in a program –

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

struct Books {
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
};

int main ( ) {

    struct Books Book1;    /* Declare Book1 of type Book */
    struct Books Book2;    /* Declare Book2 of type Book */

    /* book 1 specification */
    strcpy ( Book1.title, "C Programming" );
    strcpy ( Book1.author, "Nuha Ali" );
    strcpy ( Book1.subject, "C Programming Tutorial" );
    Book1.book_id = 6495407;
    /* book 2 specification */
    strcpy ( Book2.title, "Telecom Billing" );
    strcpy ( Book2.author, "Zara Ali" );
    strcpy ( Book2.subject, "Telecom Billing Tutorial" );
    Book2.book_id = 6495700;

    /* print Book1 info */
    printf ( "Book 1 title: %s\n", Book1.title );
    printf ( "Book 1 author: %s\n", Book1.author );
    printf ( "Book 1 subject: %s\n", Book1.subject );
    printf ( "Book 1 book_id: %d\n", Book1.book_id );

    /* print Book2 info */
    printf ( "Book 2 title: %s\n", Book2.title );
    printf ( "Book 2 author: %s\n", Book2.author );
    printf ( "Book 2 subject: %s\n", Book2.subject );
```

```
printf ( "Book 2 book_id : %d\n", Book2.book_id) ;

return 0;
}
```

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

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

## Nested Structure in C:

C provides us the feature of nesting one structure within another structure by using which, complex data types are created. For example, we may need to store the address of an entity employee in a structure. The attribute address may also have the subparts as street number, city, state, and pin code. Hence, to store the address of the employee, we need to store the address of the employee into a separate structure and nest the structure address into the structure employee. Consider the following program.

```
1. #include<stdio.h>
2. struct address
3. {
4.     char city[20];
5.     int pin;
6.     char phone[14];
7. };
8. struct employee
9. {
10.    char name[20];
11.    struct address add;
12.};
13. void main ()
14. {
15.    struct employee emp;
16.    printf("Enter employee information?\n");
17.    scanf("%s %s %d %s",emp.name,emp.add.city, &emp.add.pin, emp.add.phone);
18.    printf("Printing the employee information...\n");
19.    printf("name: %s\nCity: %s\nPincode: %d\nPhone: %s",emp.name,emp.add.city,emp
    .add.pin,emp.add.phone);
20. }
```

## Output

Enter employee information?

Arun

Delhi

110001

1234567890

Printing the employee information....

name: Arun

City: Delhi

Pincode: 110001

Phone: 1234567890

## Why use an array of structures?

Consider a case, where we need to store the data of 5 students. We can store it by using the structure as given below.

```
1. #include<stdio.h>
2. struct student
3. {
4.     char name[20];
5.     int id;
6.     float marks;
7. };
8. void main()
9. {
10.    struct student s1,s2,s3;
11.    int dummy;
12.    printf("Enter the name, id, and marks of student 1 ");
13.    scanf("%s %d %f",s1.name,&s1.id,&s1.marks);
14.    scanf("%c",&dummy);
15.    printf("Enter the name, id, and marks of student 2 ");
16.    scanf("%s %d %f",s2.name,&s2.id,&s2.marks);
```

```
17. scanf("%c",&dummy);
18. printf("Enter the name, id, and marks of student 3 ");
19. scanf("%s %d %f",s3.name,&s3.id,&s3.marks);
20. scanf("%c",&dummy);
21. printf("Printing the details....\n");
22. printf("%s %d %f\n",s1.name,s1.id,s1.marks);
23. printf("%s %d %f\n",s2.name,s2.id,s2.marks);
24. printf("%s %d %f\n",s3.name,s3.id,s3.marks);
25. }
```

## Output

```
Enter the name, id, and marks of student 1 James 90 90
Enter the name, id, and marks of student 2 Adoms 90 90
Enter the name, id, and marks of student 3 Nick 90 90
Printing the details....
James 90 90.000000
Adoms 90 90.000000
Nick 90 90.000000
```

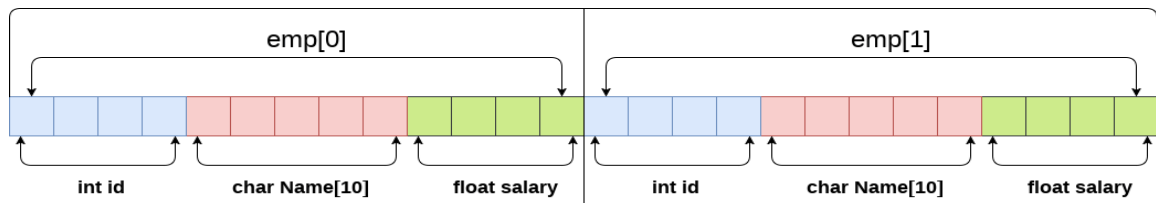
In the above program, we have stored data of 3 students in the structure. However, the complexity of the program will be increased if there are 20 students. In that case, we will have to declare 20 different structure variables and store them one by one. This will always be tough since we will have to declare a variable every time we add a student. Remembering the name of all the variables is also a very tricky task. However, C enables us to declare an array of structures by using which, we can avoid declaring the different structure variables; instead we can make a collection containing all the structures that store the information of different entities.

## Array of Structures in C

An array of structures 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



```
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`

Let's see an example of an array of structures that stores information of 5 students and prints it.

```
1. #include<stdio.h>
2. #include <string.h>
3. struct student{
4.     int rollno;
5.     char name[10];
6. };
7. int main(){
8.     int i;
9.     struct student st[5];
10. printf("Enter Records of 5 students");
11. for(i=0;i<5;i++){
12.     printf("\nEnter Rollno:");
13.     scanf("%d",&st[i].rollno);
14.     printf("\nEnter Name:");
15.     scanf("%s",&st[i].name);
16. }
17. printf("\nStudent Information List:");
18. for(i=0;i<5;i++){
19.     printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);
20. }
21. return 0;
22. }
```

### Output:

```
Enter Records of 5 students
Enter Rollno:1
```

Enter Name:Sonoo

Enter Rollno:2

Enter Name:Ratan

Enter Rollno:3

Enter Name:Vimal

Enter Rollno:4

Enter Name:James

Enter Rollno:5

Enter Name:Sarfraz

Student Information List:

Rollno:1, Name:Sonoo

Rollno:2, Name:Ratan

Rollno:3, Name:Vimal

Rollno:4, Name:James

Rollno:5, Name:Sarfraz

## Union in C:-

Like [Structures](#), union is a user defined data type. In union, all members share the same memory location.

For example in the following C program, both x and y share the same location. If we change x, we can see the changes being reflected in y.

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

```
// Declaration of union is same as structures
```

```
union test {
```

```
    int x, y;
```

```
};
```

```
int main()
```

```
{
```

```
    // A union variable t
```

```
    union test t;
```

```
t.x = 2; // t.y also gets value 2
printf("After making x = 2:\n x = %d, y = %d\n\n",
      t.x, t.y);
t.y = 10; // t.x is also updated to 10
printf("After making y = 10:\n x = %d, y = %d\n\n",
      t.x, t.y);
return 0;
}
```

**Output:**

After making x = 2:

x = 2, y = 2

After making y = 10:

x = 10, y = 10

## Difference between Structure and union:

Structure	Union
You can use a <code>struct</code> keyword to define a structure.	You can use a <code>union</code> keyword to define a union.
Every member within structure is assigned a unique memory location.	In union, a memory location is shared by all the data members.
Changing the value of one data member will not affect other data members in structure.	Changing the value of one data member will change the value of other data members in union.
It enables you to initialize several members at once.	It enables you to initialize only the first member of union.
The total size of the structure is the sum of the size of every data member.	The total size of the union is the size of the largest data member.
It is mainly used for storing various data types.	It is mainly used for storing one of the many data types that are available.
It occupies space for each and every member written in inner parameters.	It occupies space for a member having the highest size written in inner parameters.
You can retrieve any member at a time.	You can access one member at a time in the union.
It supports flexible array.	It does not support a flexible array.