# Unit-1

## Introduction to C programming

C is a general-purpose, structured programming language. Its instructions consists of terms that resemble algebraic expression, augmented by certain English keywords such as if, else, for, do and while, etc. C contains additional features that allow it to be used at a lower level, thus bridging the gap between machine language and the more conventional high level language. This flexibility allows C to be used for system programming (e.g. for writing operating systems as well as for applications programming such as for writing a program to solve mathematical equation or for writing a program to bill customers).

C has now become a widely used professional language for various reasons:

- Easy to learn
- Structured language
- It produces efficient programs
- It can handle low-level activities
- It can be compiled on a variety of computer platforms

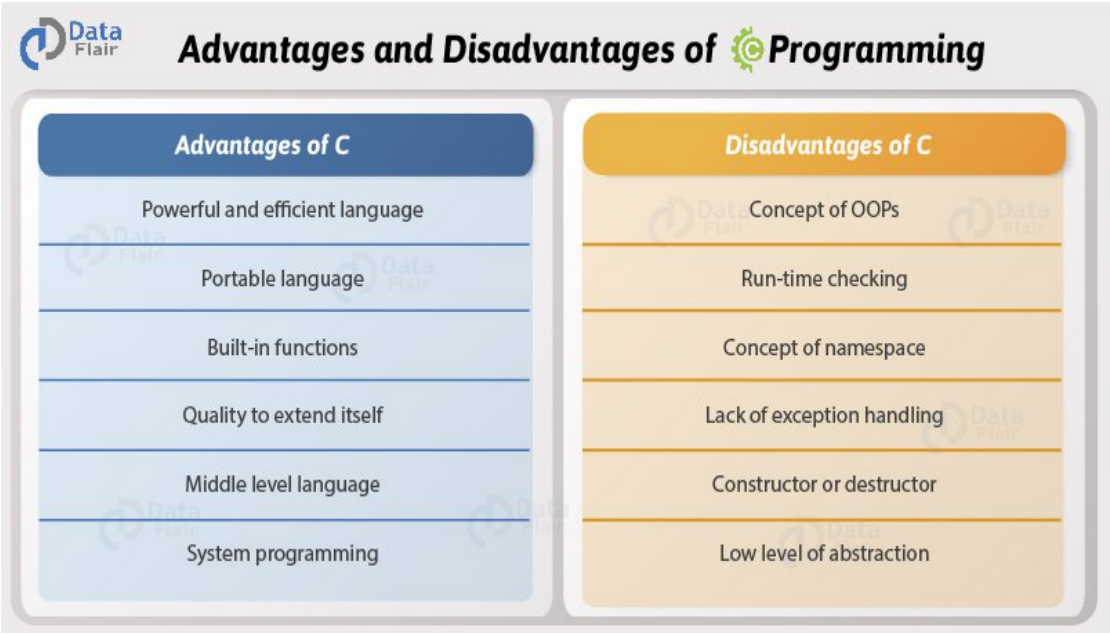
## Importance of C

Now-a-days, the popularity of C is increasing probably due to its many desirable qualities. It is a robust language whose rich set of built-in functions and operators can be used of built-in functions and operators can be used to write any complex program. The C compiler combines the capabilities of an assemble language with the features of a high-level language and therefore it well suited for writing both system software and business packages. In fact, many of the C compilers available in the market are written in C.

Programs written in C are efficient and fast. This is due to its variety of data types and powerful operators. It is many times faster than BASIC (Beginners All Purpose Symbolic Instruction Code – a high level programming language).C Language is well suited for structure programming thus requiring the user to think of a problem in terms of function modules or blocks. A proper collection of these modules would make a complete program. This modular structure makes program debugging, testing and maintenance.

Another important feature of C is its ability to extend itself. A C program is basically a collection of functions that are supported by the C library. We can continuously add our own function to the C library. With the availability of a large number of functions, the programming task becomes simple.

## Advantages and Disadvantages of C Programming




## History of C Programming

The C programming language came out of Bell Labs in the early 1970s. According to the Bell Labs paper The Development of the C Language by Dennis Ritchie, "The C programming language was devised in the early 1970s as a system implementation language for the nascent Unix operating system. Derived from the type less language BCPL, it evolved a type structure; created on a tiny machine as a tool to improve a meager programming environment." Originally, Ken Thompson, a Bell Labs employee, desired to make a programming language for the new UNIX platform. Thompson modified the BCPL system language and created B. However, not many utilities were ever written in B due to its slow nature and inability to take advantage of PDP-11 features in the operating system. This led to Ritchie improving on B, and thus creating C

## Basic Structure of C programs

Every C program consists one or more modules called function. One of the function must be called main ( ).A function is a sub-routine that may include one or more statements designed to perform a specific task. A C program may contain one or more sections shown in fig:

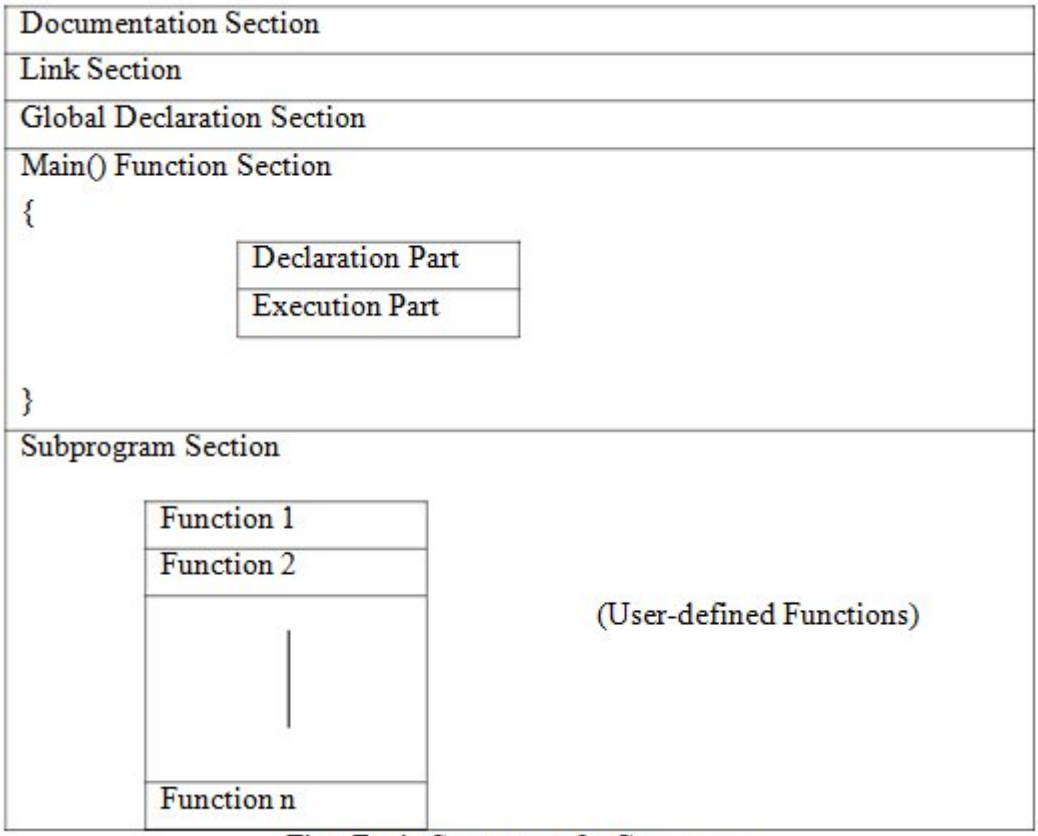


Fig : Basic Structure of a C program

**Documentation Section**

The documentation section is the part of the program where the programmer gives the details associated with the program. He usually gives the name of the program, the details of the author and other details like the time of coding and description. It gives anyone reading the code the overview of the code.

**Example**

**// program by ram (single line comment)**

/* **(multi line comment)**

Author: Manthan Naik

date: 09/08/2019

*/

**Link Section**

This part of the code is used to declare all the header files that will be used in the program. This leads to the compiler being told to link the header files to the system libraries.

**Example**

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

**Global Declaration Section**

This part of the code is the part where the global variables are declared. All the global variable used are declared in this part. The user-defined functions are also declared in this part of the code.

**Example**

```
float area(float r);
int a=7;
```

**Main Function Section**

Every C-programs need to have the main function. Each main function contains two parts. A declaration part and an Execution part. The declaration part is the part where all the variables are declared. The execution part begins with the curly brackets and ends with the curly close bracket. Both the declaration and execution part are inside the curly braces.

**Example**

```
Void main()
{
int a=10;
printf(" %d", a);
}
```

**Sub Program Section**

All the user-defined functions are defined in this section of the program.

**Example**

```
1  int add(int a, int b)
2  {
3  return a+b;
4  }
```

## Problem Analysis

If you have studied a problem statement, then you must analyze the problem and determine how to solve it. First, you should know the type of problem that is, nature of problem. In programming point of view, the problem must be computing. At first you try to solve manually. If it is solvable manually by using your idea and knowledge, then you can use such idea and principle in programming and solve the problem by using computer. So, you must have well knowledge about a problem. In order to get exact solution, you must analyze the problem. To analyze means you should try to know the steps that lead you to have an exact solution.

Suppose you know the steps to be followed for solving the given problem but while solving the problem you forget to apply some steps or you apply the calculation steps in the wrong sequences. Obviously, you will get a wrong answer. Similarly, while writing a computer program, if the programmer leaves out some of the instructions for the computer or writes the instructions in the wrong sequences, then the computer will calculate a wrong answer. Thus to produce an effective computer program, it is necessary that the programmers write each and every instruction in the proper sequence. However, the instruction sequence (logic) of a computer program can be very complex. Hence, in order to ensure that the program instructions are appropriate for the problem and are in correct sequence.

## Algorithm Development & Flowcharting:

The term algorithm may be formally defined as a sequence of instructions designed in such a way that if the instructions are executed in the specified sequence, the desired result will be obtained. An algorithm must possess the following characteristics:

1. Each and every instruction should be precise and unambiguous.
2. Each instruction should be such that it can be performed in a finite time.
3. One or more instruction should not be repeated infinitely. This ensures that the algorithm will ultimately terminate.
4. After performing the instructions, that is after the algorithm terminates, the desired results must be obtained.

**Example:**

Write an algorithm to read 2 numbers form user and display the resulting sum.

1. Start.
2. Read 2 numbers and store in variables, say A and B.
3. Store the sum of A and B in C.
4. Display the value in C.
5. Stop.

Write an algorithm to find area of circle

1. Start
2. Read the value of r
3. Set pi equal to 3.14
4. Calculate area using formula, a=pi*r*r
5. Print area
6. Stop

Write an algorithm to find largest among two inputs numbers

Step 1: Start

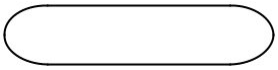
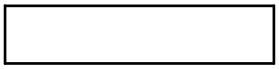
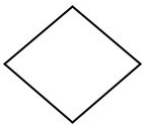
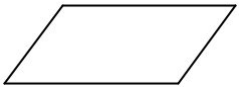
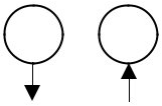
Step 2: Read a, b . /* a, b two numbers */
Step 3: If a>b then /*Checking */
Display “a is the largest number”.
Otherwise
Display “b is the largest number”.
Step 4: Stop.

**Flowchart:**

A flowchart is a pictorial representation of an algorithm that uses boxes of different shapes to denote different types of instructions. The actual instructions are written within these boxes using clear and concise statements. These boxes are connected by solid lines having arrow marks to indicate the flow of operation, that is, the exact sequence in which the instructions are to be executed.

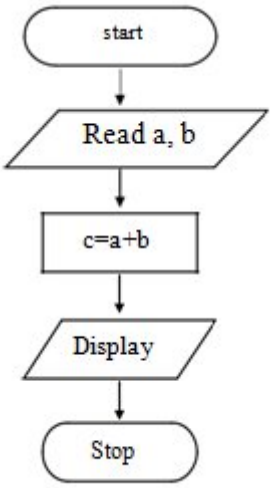
Normally, an algorithm is first represented in the form of a flowchart and the flowchart is then expressed in some programming language to prepare a computer program. The main advantage of this two steps approach in program writing is that while drawing a flowchart one is not concerned with the details of the elements of programming language. Since a flowchart shows the flow of operations in pictorial form, any error in the logic of the procedure can be detected more easily than in the case of a program. Once the flowchart is ready, the programmer can forget about the logic and can concentrate only on coding the operations in each box of the flowchart in terms of the statements of the programming language. This will normally ensure an error-free program.

**Basic blocks used for drawing flowcharts:**

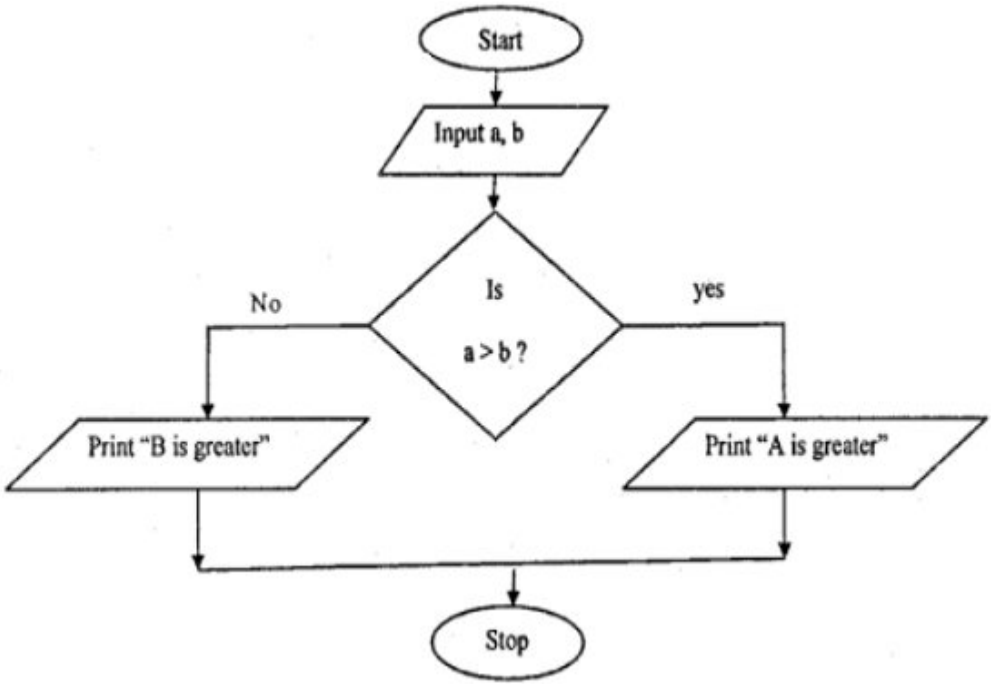
| Structure | Purpose |
|---|---|
|  | Start / Stop |
|  | Processing |
|  | Decision making |
|  | Input / Outputs |
|  | Connector |

**Examples:**

Draw a flowchart for read 2 numbers form user and display the resulting sum.



Draw a flowchart for largest among two input numbers




Draw a flowchart for calculate area of circle

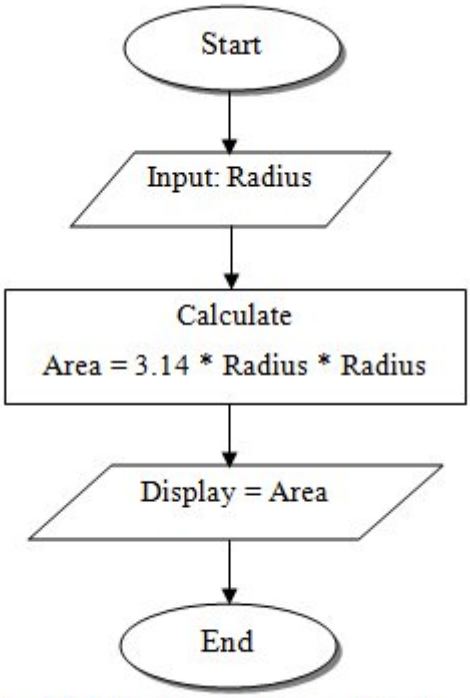



Fig. Flowchart to print Area of Circle

## Writing, Compiling, debugging, executing and testing a C program

Writing any C programs includes following major steps:

- **Program Design**
- **Program Coding**
- **Compilation & Execution**
- **Program testing and debugging**

### Program Design

Program design is the foundation that is at the heart of program development cycle. Before getting your hands on the source code, it is crucial to understand the program from all sides and develop program development strategy.

You can break down program design into four steps:

- Problem Analysis
- Generating the program structure or blue print
- Algorithm& Flowchart development of the program
- Proper selection of control statements

### Program Coding

Once you have developed the full blue print of the program, it's time to write source code according to the algorithm. That means be ready to translate your flowchart into a set of instruction. It's time to use your knowledge of C programming to work and try to write your code as simple as you can.

Following are the elements of program coding:

- Proper documentation or commenting
- Proper way for construction of statements
- Clear input/output formats
- Generality of the program

**Compilation & Execution**

The process by which source codes of a computer (programming) language are translated into machine codes is known as compilation. After compilation if everything is ok, the code is going under other process that is known as execution. We can get the required output after execution process.

**Debugging & Testing:**

The process of finding and removing errors (also sometimes called buggs) from a program is known as debugging. One simple method of debugging is to place print statements throughout the program to display the values of variables. It displays the dynamics of a program and allows us to examine and compare the information at various points. Once the location of an error is identified and the error is corrected, the debugging statements may be removed.

Generally programmers commit three types of errors. They are

1. Syntax errors
2. Logic errors
3. Run-time errors

Syntax errors are those errors which are raised from violating the rules of programming language. On encountering these errors, a computer displays error message. It is easy to debug. Logic errors are those which raised when programmers proceed the logic process in wrong way or miss the some statements. It is difficult to debug such errors because the computer does not display them. Run-time errors are those which occur when programmers attempt to run ambiguous instructions. They occur due to infinite loop statement, device errors, software errors, etc. The computer will print the error message .Some of runtime errors are:

Testing is the process of reviewing and executing a program with the intent of detecting errors. Testing can be done manually and computer based testing. Manual Testing is an effecting error-detection process and is done before the computer based testing begins. Manual testing includes code inspection by the programmer, code inspection by a test group and a review by a peer group. Computer based testing is done by computer with the help of compiler (a program that changes source codes into machine codes word by word).

# Unit- 2 Elements of C

## C Tokens

In a passage of text, individual words and punctuation marks are called tokens. Similarly, in a C program the smallest individual units are also known as C tokens. C has six types of tokens:

1. Identifiers e.g.: x area Sum
2. Keywords e.g.: int float for While
3. Constants e.g.: -15.5 100 3.14
4. Strings e.g.: “ABC” “year”
5. Operators e.g.: + - *
6. Special Symbols e.g.: ( ) [ ] { }

### Character Set:

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

**+ - * / = % & # ! ? ^ “ ‘ \ () [] {}: ; . ,**

### Identifiers & Keywords:

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

Rules for Identifier:

1. First character must be an alphabet (or Underscore).
2. Must consist of only letters, digits or underscore.
3. Only first 31 characters are significant.
4. Cannot use a keyword.
5. Must not contain white space.

**The following names are valid identifiers:**

X a12 sum_1 _temp name area tax_rate TABLE

**The following names are not valid identifier**

| | |
|---|---|
| $4^{th}$ | The first character must be letter |
| "x" | Illegal characters (") |
| Order-no | Illegal character (-) |
| Error flag | Illegal character (blank space) |

**Keywords:**

There are certain reserved words, called keywords that have standard, Predefined meanings in C. These keywords can be used only for their intended purpose; they cannot be used as programmer-defined identifiers. The standard keywords are

| | | | | |
|---|---|---|---|---|
| Auto | break | case | char | const |
| Continue | default | do | double | else |
| Enum | extern | float | for | goto |
| If | int | long | register | return |
| Short | signed | sizeof | static | struct |
| Switch | typedef | union | unsigned | void |
| Volatile | while | | | |

**Data Types:**

Data type is means to identify the type of data and associated operation for handling it. C language is rich in its data types. C supports several different types of data, each of which may be represented differently within the computer memory. There are three cases of data types:

1. Basic data types (Primary or Fundamental) e.g.: int, char
2. Derived data types e.g.: array, pointer, function
3. User defined data types e.g.: structure, union, enum

The basic data types are also known as built in data types. The basic data types are listed below. Typical memory requirements are also given:

| Data Types | Description | Typical Memory Requirement |
|---|---|---|
| Char | single character | 1 byte |
| Int | integer quantity | 2 bytes |
| Float | floating-point number | 4 bytes |
| Double | double-precision floating point number | 8 bytes |

In order to provide some control over the range of numbers and storage space, C has following classes: signed, unsigned, short, long.

| Types | Size |
|---|---|
| char or signed char | 1 byte |
| unsigned char | 1 byte |
| int | 2 bytes |
| short int | 1 byte |
| unsigned short int | 1 byte |
| signed int | 2 bytes |
| unsigned int | 2 bytes |
| long int | 4 bytes |
| signed long int | 4 bytes |
| unsigned long int | 4 bytes |
| float | 4 bytes |
| double | 8 bytes |
| long double | 10 bytes |

Void is also a built-in data type used to specify the type of function. The void type has no values.

**Constants &Variables**

Constants in C refer to fixed values that do not change during the execution of a program. There are four basic types of constants in C. They are integer constants, floating point constants, character constants and string constants.

Integer and floating point constants represent numbers. They are often referred to collectively as numeric _ type constants. The following rules apply to all numeric type constants.

- Commas and blank spaces cannot be included within the constants.
- The constant can be preceded by a minus (-) if desired. The minus sign is an operator that changes the sign of a positive constant though it can be thought of as a part of the constant itself.
- The value of a constant cannot exceed specified minimum and maximum bounds. For each type of constant, these bounds will vary from one C compiler to another.

**Variables**

In programming, a variable is a container (storage area) to hold data. To indicate the storage area, each variable should be given a unique name identifier. Variable names are just the symbolic representation of a memory location. For example:

```
int playerScore = 95;
```

Here, *playerScore* is a variable of `int` type. Here, the variable is assigned an integer value `95`. The value of a variable can be changed, hence the name variable.

```
char ch = 'a';
// same code
ch = 'b';
```

**Rules for Naming Variables**

- The first letter of a variable name must be alphabet (underscore is also allowed).
- Variable can only contain letters, digits, and underscores.
- White space is not allowed.
- Keywords cannot be used as an identifier to name variables.
- Since C is case sensitive language variable name written in uppercase will differ from lowercase.

**Escape Sequence:**

Certain nonprinting character, as well as the backslash (\) and apostrophe (`), can be expressed in terms of escape sequences. An escape sequence always begins with a backslash and is followed by one or more special characters. For example, a linefeed (LF), which is

referred to as a newline in C, can be represented as \n. Such escape sequences always represent single characters, even though they are written in terms of two or more characters.

**The commonly used escape sequences are listed below:**

| Character | Escape Sequence |
|---|---|
| backspace | \b |
| horizontal tab | \t |
| vertical tab | \v |
| newline (line feed) | \n |
| form feed | \f |
| carriage return | \r |
| quotation mark (”) | \” |
| apostrophe ( `) | \` |
| question mark (?) | \? |
| backslash (\) | \\ |
| Null | \0 |

**Statements and comments**

C programs are collection of Statements, statements is an executable part of the program it will do some action. In general all arithmetic actions and logical actions are falls under Statements categories. There are few Statement categories

- Expression Statements.
- Compound Statements.
- Selection Statements.
- Iterative Statements.
- Jump Statements

## Comments

A **comment** is an explanation or description of the source code of the program. It helps a developer explain logic of the code and improves program readability. At run-time, a comment is ignored by the compiler. In the C Programming Language, you can place comments in your source code that are not executed as part of the program.

Comments provide clarity to the C source code allowing others to better understand what the code was intended to accomplish and greatly helping in debugging the code. Comments are especially important in large projects containing hundreds or thousands of lines of source code or in projects in which many contributors are working on the source code. Comments can span several lines within your C program. Comments are typically added directly above the related C source code.

Adding source code comments to your C source code is a highly recommended practice. In general, it is always better to over comment C source code than to not add enough.

**There are 2 types of comments in the C language.**

1. Single Line Comments
2. Multi-Line Comments

**Single Line Comments**

Single line comments are represented by double slash \\. Let's see an example of a single line comment in C.

```
#include<stdio.h>

#include<conio.h>

Void main()

{

    Printf ("Hello C");        //printing information

getch();

 }
```

**Multi-Line Comments**

Multi-Line comments are represented by slash asterisk * ... *\. It can occupy many lines of code, but it can't be nested. Let’s see an example of multiline comments in C.

```
#include<stdio.h>
#include<conio.h>
Void main ()
{
   printf("Hello C");                /*printing information
    getch();                              Multi-Line Comment*/
}
```

**Delimiters**

A delimiter is one or more characters that separate text strings. Common delimiters are commas (,), semicolon (;), quotes (" “), braces ({}) or slashes (/ \). When a program stores sequential or tabular data, it delimits each item of data with a predefined character.

## Unit 3 Input and Output

A program is a set of instructions that takes some data and provides some data after execution. The data that is given to a program is known as input data. Similarly, the data that is provided by a program is known as output data. Generally, input data is given to a program from a keyboard (a standard input device) or a file. The program then proceeds the input data and the result is displayed on the screen (monitor – a standard output device) or a file. Reading input data from keyboard and displaying the output data on screen, such input output system is considered as conio input out.

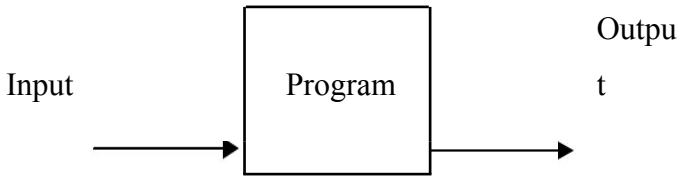



Fig: Input/ Output

To perform input/output operation in console mode, C has a number of input and output functions. When a program needs data, it takes the data through the input functions and sends the results to output devices through the output functions. Thus the input/output functions are the link between the user and the terminal.

As keyboard is a standard input device, the input functions used to read data from keyboard are called standard input functions. The standard input functions are scanf( ), getchar( ), getch( ), gets( ), etc. Similarly, the output functions which are used to display the result on the screen are called standard output functions. The standard output functions are printf( ), putchar(), putch( ), puts( ), etc. The standard library stdio.h provides functions for input and output. The instruction #include<stdio.h> tells the compiler to search for a file named stdio.h and place its contents at this point in the program. The contents of the header file become part of the source code when it is compiled.

## Input/ Output Management

In this topic, you will learn to use scanf() function to take input from the user, and printf() function to display output to the user.

## C Output

In C programming, `printf()` is one of the main output function. The function sends formatted output to the screen. For example,

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

**Output**

```
Number = 5
```

We use %d format specifier to print int types. Here, the %d inside the quotations will be replaced by the value of *testInteger*.

## C Input

In C programming, `scanf()` is one of the commonly used function to take input from the user. The `scanf()` function reads formatted input from the standard input such as keyboards.

For example,

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

```
int testInteger;
printf("Enter an integer: ");
scanf("%d", &testInteger);
printf("Number = %d",testInteger);
getch ();
}
```

**Output**

```
Enter an integer: 4
Number = 4
```

Here, we have used `%d` format specifier inside the `scanf()` function to take `int` input from the user. When the user enters an integer, it is stored in the *testInteger* variable
(Notice, that we have used `&testInteger` inside `scanf()`. It is because *&testInteger* gets the address of *testInteger*, and the value entered by the user is stored in that address)

## Conversion specifiers

The format or conversion specifiers are used in C for input and output purposes. Using this concept the compiler can understand that what type of data is in a variable during taking input using the scanf() function and printing using printf() function. Format specifiers start with a percentage *%* operator and followed by a special character for identifying the type of data.

**Here is a list of format or conversion specifiers**

| Format Specifier | Type |
|---|---|
| %c | Character |
| %d | Integer |
| %f | Float values |
| %ld | Long integer |
| %lf | Double |
| %p | Pointer |
| %s | String |
| %u | Unsigned integer |

**Some examples program**

/*Write a program to add two numbers*/

```
#include<stdio.h>
#include<conio.h>
Void main ()
{
int num1,num2,result;
printf ("enter first number:");
scanf ("%d",&num1);
printf ("enter second number:");
scanf ("%d",&num2);
result = num1+num2;
```

/*Write a program to display area of circle*/

```
#include<stdio.h>
 #include<conio.h>
 #define pi 3.14
 Void main ()
 {
 int r;
float area;
printf("enter the value of radius:");
scanf("%d",&r);
area=pi*r*r;
```

```
printf ("the sum is:%d", result);
getch();
}
```

```
printf("the area of circle is:%f",area);
 getch();
}
```

/*Write a program to calculate simple interest*/

```
#include<stdio.h>
#include<conio.h>
Void main ()
int p ,t ,r , i;
printf("enter the priniple amount");
scanf("%d", &p);
printf("enter rate of interest");
scanf("%d", &r);
printf("enter time");
scanf("%d", &t);
i= (p*t*r)/100;
printf("The interest is:%d", i);
getch ();
}
```

/*Write a program to enter any character and display it to the screen*/

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

```
Char text;
Printf("enter any character:");
Scanf("%c", &text);
Printf("The entered character is :%c",text);
getch();
}
```

**Types of I/O**

The input/output functions are classified into two types:

i. Formatted functions

ii. Unformatted functions

**Formatted Functions:**

Formatted functions allow the input read from the keyboard or the output displayed on screen to be formatted according to our requirements. The input function scanf() and output function: printf() fall under this category. While displaying a certain data on screen, we can specify the number of digits after decimal point, number of spaces before the data, the position where the output is to be displayed, etc, using formatted functions.

**Formatted Input:**

Formatted input refers to an input data that has been arranged in a particular format. For example, consider the following data.

20 11.23 Ram

The above line contains three types of data and must be read according to its format. The first be read into a variable int, the second into float, and the third into char. This is possible in C using the scanf function. scanf( )stands for scan formatted.

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

scanf (“control string”, arg1, arg2, …….., argn)

where,

control string refers to a string containing certain required formatting information so also known as format string and arg1, arg2, ……, argn are arguments that represent the individual input data items. Actually, the arguments represent pointers that indicate the addresses of the data items within the computer’s memory.

The control string consists of individual groups of characters, with one character group for each input data item. Each character group must begin with a percent sign (%). In its simplest form, a single character group will consist of the percentage sign, followed by a conversion character which indicates the types of corresponding data item. Within the control string, multiple character groups can be contiguous, or they can be separated by whitespace (i.e. blankspace), tabs or newline characters. If whitespace characters are used to separate multiple character groups in the control string, then all consecutive white-space characters in the input data will be read but ignored. The use of blank spaces as character group separators is very common.

**Example 1**

```
#inlcude<stdio.h>
  Void main( )
        {
        char item[20] ;
        int number ;
        float cost ;
                        - -----
                scanf (“%s %d %f”, item, &number, &cost) ;
                        -------
                                        }
```

The following data items could be entered from the standard input device when the program is executed.

Dharan  1245  0.05  ↵

**Example: 2**

```
#include<stdio.h>
  main( )
  {
   int a, b, c ;
   -----
   scanf (“%3d %3d %3d”, &a, &b, &c) ;
   -----
   }
```

Suppose the input data items that are entered as
123↵

Then the following assignment will result:

a = 1, b = 2, c = 3

If the data had been entered as

123 456 789

Then the assignment would be

a = 123, b = 456, c = 789

Now suppose that the data had been entered as
123456789

Then the assignments would be

a = 123, b = 456, c= 789

Finally, suppose that the data had been entered as

1234 5678 9

The resulting assignments would now be

a = 123 b = 4 c= 567

**Example: 3**

```
#include<stdio.h>
    main( )
{
    char item[20] ;
    int number ;
    float cost

    ------

    scanf ("%s %*d %f", item, &number, &cost),

    ------

}
```

If the corresponding data item are input

ram 12345 0.05

ram is assigned to item and 0.05 will be assigned to cost. However 12345 will not be assigned number because of asterisk, which is interpreted as an assignment suppression character.

**Formatted Output:**

Formatted output refers to the output of data that has been arranged in a particular format. The printf() is a built-in function which is used to output data from the computer onto a standard output device i.e. screen, This function can be used to output any combination of numerical values, single character and strings. The printf() statement provides certain features that can be

used to control the alignment and spacing of print-outs on the terminals. The general form of printf() statement is:

printf (control string, arg1, arg2, …. , argn)

where,

Control string refers to a string that contains formatting information, and arg1, arg2, …….,argn are arguments that represent the individual output data item. The arguments can be written as constants, single variable or array names, or more complex expressions.

For example:

```
#include<stdio.h>

    Void main( )

    {

      char item [20] ;

      int number ;

      float cost ;

      ------

      printf (“%s%d%f ”, item, number, cost) ;

    }
```

Suppose ram, 12345 and 0.5 have been assigned to item, number and cost. So, the output generated will be ram123450.5

**Format for Integer Output:**

%wd

Where w is the integer number specifying the minimum field width of output data. If the length of the variable is less than the specified field width, then the variable is right justified with leading blanks. If the specified field width is less than length of variable than fill all variable.

For example: integer number

i.e. int n = 1234

| Format | Output |
|---|---|
| printf (“%d”, n) ; | 1 2 3 4 |
| printf (“%6d”, n) | _ _ 1 2 3 4 |
| printf (“%2d”, n) | 1 2 3 4 |
| printf (“%-6d”, n) | 1 2 3 4 _ _ |
| printf (“%06d”, n) | 0 0 1 2 3 4 |

**Format for floating point output**:

The general form:

%w.pf

%w.pe

Where, w is the integer width including decimal point p
is the precision f and e are conversion characters

**Example:**

float x = 12.3456

| | |
|---|---|
| printf (“%7.4f”, x) ; | 1 2 . 3 4 5 6 |
| printf (“%7.2f”, x) ; | _ _ 1 2 . 3 5 |
| printf (“%-7.2f”, x) ; | 1 2 . 3 5 _ _ |
| printf (“%7.2f”, -x) ; | _ – 1 2 . 3 5 |

**Example:**

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int num1 = 12345;
        long num2 = 987654;
```

```
    float num3 = 98.7654;
    clrscr();
    printf("%d\n", num1);
    printf("%10d\n", num1);
    printf("%010d\n", num1);
    printf("%-10d\n", num1);
    printf("%10ld\n", num2);
    printf("%10ld\n", -num2);
    printf("%7.4f\n", num3);
    printf("%f\n", num3);
    printf("%7.2f\n", num3);
    printf("%-7.2f\n", num3);
    printf("%07.2f\n", num3);
   getch();
}
```

**Unformatted Functions**

Unformatted functions do not allow the user to read or display data in desired format. These library functions basically deal with a single character or a string of characters. The functons getchar( ), putchar( ), gets( ), puts( ), getch( ), getche( ), putch( ) are considered as unformatted functions.

**getchar( ) and putchar ():**

The getchar( ) function reads a character from a standard input device. **The general syntax is**

```
character_variable = getchar( );
```

Where character_variable is a valid C char type variable. When this statement is encountered, the computer waits until a key is pressed and assign this character to character_variable.

The putchar( ) function displays a character to the standard output device. **The general syntax of putchar( ) function is**

```
Putchar (character_variable)
```

where character_variable is a char type variable containing a character

Example:

```
#include<stdio.h>
#include<conio.h>
Void main( )
{
  char gender ;
  printf (“Enter gender M or F :” ) ;
  gender= getchar();
  printf(“The entered gender is:”);
  putchar (gender) ;
  getch( );

}
```

getch( ), getche( ) and putch( ):

The functions getch( ) and getche( ) reads a single character the instant it is typed without waiting for the enter key to be hit. The difference between them is that getch( ) reads the character typed without echoing it on the screen, while getche( ) reads the character and echoes (displays) it on the screen.

**The general syntax of getch( ):**

```
character_variable = getch( ) ;
```

**Similarly, the syntax of getche( ) is**

```
character_variable = getche( ) ;
```

The putch( ) function prints a character onto the screen. The general syntax is

```
putch (character_variable) ;
```

**Example**:

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

```
#include<conio.h>
 Void main( )
      {
        char ch1, ch2 ;
        clrscr( );
        printf (“Enter first character:”) ;
        ch1 = getch( ) ;
        printf (“\n Enter second character:”) ;
        ch2 = getche( )
        printf (“\n First character:”) ;
        putch(ch1) ;
        printf(“\n Second character:”) ;
        putch(ch2) ;
        getch();
      }
```

**gets( ) and puts( ):**

The gets( ) function is used to read a string of text containing whitespaces, until a newline character is encountered. It offers an alternative function of scanf( ) function for reading strings. Unlike scanf( ) function, it doesnot skip whitespaces.

**The general syntax of gets( ) is**

gets (string_variable) ;

The puts( ) function is used to display the string onto the terminal.

**The general syntax of puts( ) is**

puts (string _variable)

This prints the string value of string_variable and then moves the cursor to the beginning of the next line on the screen.

**Example:**

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

```
#include<conio.h>
 void main( )

        {
          char name[20] ;
          clrscr( );
          printf (“Enter your name:”) ;
          gets (name) ;
          printf (“your name is:”) ;
          puts (name) ;
          getch ();
        }
```

## Unit-4: Operators and Expression

### Operators

Operators are the foundation of any programming language. Thus the functionality of C programming language is incomplete without the use of operators. We can define operators as symbols that help us to perform specific mathematical and logical computations on operands. In other words, we can say that an operator operates the operands.
For example, consider the below statement:

```
c = a + b;
```

Here, '+' is the operator known as *addition operator* and 'a' and 'b' are operands. The addition operator tells the compiler to add both of the operands 'a' and 'b'

**C has many built-in operator types and they are classified as follows:**

**Arithmetic Operators**

An arithmetic operator performs mathematical operations such as addition, subtraction, multiplication, division etc on numerical values (constants and variables). The basic arithmetic operators are:

| Operator | Meaning of Operator |
|---|---|
| + | addition or unary plus |
| - | subtraction or unary minus |
| * | Multiplication |
| / | Division |
| % | remainder after division (modulo division) |

**Example:**

```
// Working of arithmetic operators
#include<stdio.h>
#include<conio.h>
Void main()
{
int a = 9,b = 4;
   float c;
   c = a+b;
printf("a+b = %f \n",c);
   c = a-b;
printf("a-b = %f \n",c);
   c = a*b;
printf("a*b = %f \n",c);
   c = a/b;
printf("a/b = %f \n",c);
   c = a%b;
printf("Remainder when a divided by b = %f \n",c);
   getch();
   }
```

## Output

```
a+b = 13
a-b = 5
a*b = 36
a/b = 2.25
Remainder when a divided by b=1
```

**Increment and Decrement Operators**

C programming has two operators increment ++ and decrement -- to change the value of an operand (constant or variable) by 1.

Increment ++ increases the value by 1 whereas decrement -- decreases the value by 1. These two operators are unary operators, meaning they only operate on a single operand.

**Example:**

```
// Working of increment and decrement operators
#include<stdio.h>
#include <conio.h>
Void main()
{
int a = 10, b = 100;
float c = 10.5, d = 100.5;
printf("++a = %d \n", ++a);
printf("--b = %d \n", --b);
printf("++c = %f \n", ++c);
printf("--d = %f \n", --d);
getch();
}
```

**Output**

```
++a = 11
--b = 99
++c = 11.5
--d = 99.5
```

(Note: Here, the operators ++ and -- are used as prefixes. These two operators can also be used as postfixes like a++ and a--)

**++ and - - operator as prefix and postfix**

- If you use the ++ operator as prefix like: ++var. The value of var is incremented by 1 then, it returns the value.
- If you use the ++ operator as postfix like: var++. The original value of var is returned first then, var is incremented by 1.

The -- operator works in a similar way like the ++ operator except it decreases the value by 1.

**Let's see the use of ++ as prefix and postfix in C**:

**Example:**

```
#include <stdio.h>
#include <conio.h>
void main()
 {
 int var1 = 5, var2 = 5;
 printf("%d\n", var1++);      // var1 is displayed
  printf(“%d\n”, var1);        // Then, var1 is increased to 6.
  printf("%d\n", ++var2);    // var2 is increased and then display
   getch();
}
```

Output

```
5
6
6
```

**Assignment Operators**

An assignment operator is used for assigning a value to a variable. The most common assignment operator is =.

Some of assignment operator used in c is as follows:

| Operator | Example | Same as |
|---|---|---|
| = | a = b | a = b |
| += | a += b | a = a+b |
| - = | a - = b | a = a-b |
| *= | a *= b | a = a*b |
| /= | a /= b | a = a/b |
| %= | a %= b | a = a%b |

**Example:**

```
// Working of assignment operators
#include<stdio.h>
#include <conio.h>
Void main()
{
int a = 5, c;
    c = a;
printf("c = %d\n", c);
    c += a;
printf("c = %d\n", c);
    c - = a;
printf("c = %d\n", c);
    c *= a;
printf("c = %d\n", c);
    c /= a;
```

```
printf("c = %d\n", c);
   c %= a;
printf("c = %d\n", c);
  getch();
}
```

**Output**

```
c = 5
c = 10
c = 5
c = 25
c = 5
c = 0
```

**Relational Operators**

A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0. Relational operators are used in **decision making** and **loops**.

Some of relational operator and their example are:

| Operator | Meaning of Operator | Example |
|---|---|---|
| = = | Equal to | 5 = = 3 is evaluated to 0 |
| > | Greater than | 5 > 3 is evaluated to 1 |
| < | Less than | 5 < 3 is evaluated to 0 |
| != | Not equal to | 5 != 3 is evaluated to 1 |

| Operator | Meaning of Operator | Example |
|---|---|---|
| >= | Greater than or equal to | 5 >= 3 is evaluated to 1 |
| <= | Less than or equal to | 5 <= 3 is evaluated to 0 |

**Example**

```
// Working of relational operators
#include<stdio.h>
#include <conio.h>
Void main()
{
int a = 5, b = 5, c = 10;
printf("%d = = %d is %d \n", a, b, a = = b);
printf("%d = = %d is %d \n", a, c, a = = c);
printf("%d > %d is %d \n", a, b, a > b);
printf("%d > %d is %d \n", a, c, a > c);
printf("%d < %d is %d \n", a, b, a < b);
printf("%d < %d is %d \n", a, c, a < c);
printf("%d != %d is %d \n", a, b, a != b);
printf("%d != %d is %d \n", a, c, a != c);
printf("%d >= %d is %d \n", a, b, a >= b);
printf("%d >= %d is %d \n", a, c, a >= c);
printf("%d <= %d is %d \n", a, b, a <= b);
printf("%d <= %d is %d \n", a, c, a <= c);
getch();
}
```

**Output**

```
5 == 5 is 1
5 == 10 is 0
```

5 > 5 is 0

5 > 10 is 0

5 < 5 is 0

5 < 10 is 1

5 != 5 is 0

5 != 10 is 1

5 >= 5 is 1

5 >= 10 is 0

5 <= 5 is 1

5 <= 10 is 1

**Logical Operators**

An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false. Logical operators are commonly used in **decision making in C programming.**

| Operator | Meaning | Example |
|---|---|---|
| && | Logical AND. True only if all operands are true | If c = 5 and d = 2 then, expression `((c==5) && (d>5))` equals to 0. |
| \|\| | Logical OR. True only if either one operand is true | If c = 5 and d = 2 then, expression `((c==5) \|\| (d>5))` equals to 1. |
| ! | Logical NOT. True only if the operand is 0 | If c = 5 then, expression `!(c==5)` equals to 0. |

**// Working of logical operators**

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

```
int a = 5, b = 5, c = 10, result;

   result = (a == b) && (c > b);
printf("(a == b) && (c > b) is %d \n", result);

   result = (a == b) && (c < b);
printf("(a == b) && (c < b) is %d \n", result);

   result = (a == b) || (c < b);
printf("(a == b) || (c < b) is %d \n", result);

   result = (a != b) || (c < b);
printf("(a != b) || (c < b) is %d \n", result);

}
```

**Output**

```
(a == b) && (c > b) is 1
(a == b) && (c < b) is 0
(a == b) || (c < b) is 1
(a != b) || (c < b) is 0
```

**Conditional Operators:**

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

expression1 ? expression2 **:** expression3

When evaluating a conditional expression, expression1 is evaluated first. If expression1 is true, the value of expression2 is the value of conditional expression. If expression1 is false, the value of expression3 is the value of conditional expression. For example:

```
a = 10 ;
b = 15 ;
x = (a > b) ? a : b ;
```

In this example, x will be assigned the value of b.

# Unit 5 Control Statement

The statements which alter the flow of execution of the program are known as control statement. In the absence of control statements, the instruction or statements are executed in the same order in which they appear in the program. Sometimes, we may want to execute some statements several times. Sometime we want to use a condition for executing only a part of program. So, control statements enable us to specify the order in which various instructions in the program is to be executed.

**There are two types of control Statement:**

1. **Decisions/Selection**: if, if...else, else if , nested if....else, switch
2. **Loops/iterative** : for, while, do-while

**Decisions**

Decision making or selection statements control the flow of execution; they fall under the category of control statements. Decision making or selection statement structure requires that the programmer specifies one or more conditions to be evaluated or tested by the program along with statements to be executed, if the condition is determined to be true &optionally, if the condition is false other statements will be executed.

**Following are decision making/Selection statements:**

- if statements
- if....else statements
- else if statement
- Nested if...else statement
- switch statement

**if statement:**

If statement is a powerful decision making statement and is used to control the flow of execution of statements. This is a bi-directional condition control statements. This statement is used to test a condition and take one of two possible actions, If the condition is true then a single statement or a block of statements is executed (one part of the program), otherwise another single statement or a block of statements is executed (other part of the program). In C, any non-zero value is regarded as true while zero is regarded as false.

Syntax:

```
if (condition)
        statement1;
```

```
if (condition)
{       statement1;
        -----
        Statement n;
}
```

There can be a single statement or a block of statements after the if part.

For eg:

```
/* Program to check whether the number is –ve or not *\
  #include<stdio.h>
  #include<conio.h>
   Void main ( )
      {
        int num ;
        clrscr( );
        printf (“Enter a number to be tested:” ) ;
        scanf (“%d”, &num) ;
        if (num<0)
```

```
    {
     printf("The number is negative") ;
    }
      printf ("value of num is : %d\n", num) ;
      getch( ) ;
  }
```

**if...else statement:**

The if...else statement is an extension of the simple if statement. It is used when there are two possible actions – one when a condition is true and the other when it is false. The syntax is:

```
if (condition)
  statement1;
  else
     statement2;
```

```
if (condition)
{
        statement ;
}       ----

else

{

        Statement;

        ----

}
```

```
/* Program to check whether the number is even or odd */
#include<stdio.h>
#include<conio.h>
 Void main ( )
      {
```

```
int num, remainder ;
clrscr( );
printf (“Enter a number:”) ;
scanf (“%d”, &num) ;
remainder = num%2                    /*modular division
if (remainder = = 0)                   test for even*/
        printf (“Number is even\n”) ;
else
         printf (“ Number is odd\n”) ;

getch( );

}
```

**else if 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. The else…if statement is useful when you need to check multiple conditions within the program.

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 else 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:**

```
if(condition1)
{
//These statements would execute if the condition1 is true
}
elseif(condition2)
```

```
{
//These statements would execute if the condition2 is true
}
elseif(condition3)
{
//These statements would execute if the condition3 is true
}
.
.
else
{
//These statements would execute if all the conditions return false.
}
```

/* Program to find out the grade of a student when the marks of 4 subjects are given. The method of assuming grade is as

| | |
|---|---|
| per>=80 | grade = A |
| per<80 and per>=60 | grade = B |
| per<60 and per>=50 | grade = C |
| per<50 and per>=40 | grade = D |
| per<40 | grade = F |

Here Per is percentage

*/

```
#include<stdio.h>
#include<conio.h>
 Void main( )
        {
                float m1, m2, m3, m4, total, per ;
                char grade ;
                clrscr( );
                printf (“Enter marks of 4 subjects : ”) ;
                scanf  (“%f%f%f%f”,&m1,  &m2,  &m3,  &m4)  ;
                total = m1+m2+m3+m4 ;
```

```
        per = total /4 ;
        if(per>=80)
                grade = 'A' ;
        elseif(per>=60)
                grade = 'B' ;
        elseif(per>=50)
                grade = 'C' ;
        elseif(per>=40)
                grade = 'D' ;
        else
                grade = 'F' ;
        printf("Percentage is %f\n Grade is %c\n", per, grade) ;
        getch( );
}
```

**Nested if ...else statement**:

We can have another if... else statement in the if block or the else block. This is called nested if…else statement.

**Syntax**:

```
if(condition1)
{
        if(condition2)
          statementA1;
        else
          statement A2;
}
else
```

```
    {
        if(condition3)
          statementB1;
        else
          statementB2;
    }
```

/* Program to find largest number from three given number */

```
#include<stdio.h>
#include<conio.h>
Void main( )
    {
        int a, b, c, large ;
        clrscr( );
        printf (“Enter three numbers : ”) ;
        scanf (“%d%d%d”, &a, &b, &c) ;
        if (a>b)
          {
            if (a>c)
                large = a ;
            else
                large = c ;
          }
          else
          {
            if (b>c)
                large = b ;

            else
```

```
                    large = c ;
    }

    printf ("Largest number is %d\n", large) ;
    getch( );
}
```

**Switch Statement**

Switch statement is used for multiple choice or selection. It is used as a substitute of if- else if- else statements. It is used when multiple choices are given & one choice is to be selected.

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:**

```
switch(expression)
 {
   case constant-expression1  :
      statement(s);
      break;
   case constant-expression2  :
      statement(s);
      break;
    ……….
    ……..
   case constant-expressionN
        statement(s);
        break;
  default :
   statement(s);
}
```

```
/* Program to understand the switch control statement */
#include<stdio.h>
#include <conio.h>
Void main ()
{char grade;
  printf(“enter any character:”);
  sacnf(“%c”,&grade);
  switch(grade)
{
case'a':
        printf("Excellent\n");
break;
case'b':
         printf (“very good\n”):
         break;
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 );
  getch();
}
```

**Loops**

You may encounter situations, when a block of code needs to be executed several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. Programming languages provide various control structures that allow for more complicated execution paths.

A loop is used for executing a block of statements repeatedly until a given condition returns false. A **Loop** executes the sequence of statements many times until the stated condition becomes false. A loop consists of two parts, a body of a loop and a control statement. The control statement is a combination of some conditions that direct the body of the loop to execute until the specified condition becomes false. The purpose of the loop is to repeat the same code a number of times.

**C programming language provides the following types of loops to handle looping requirements**:

1. while loop
2. do-while loop
3. for loop
4. nested loop

**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.

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.

/* WAP to print 10 to 20 using while loop*/

```
#include <stdio.h>
#include <conio.h>
Void main ()
{
  int a = 10;
  while( a < =20 )         /* while loop execution */
 {
     printf("value of a: %d\n", a);
     a++;
  }
  getch();
}
```

**do-while loop**

Unlike **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.

```
/* program to print the number from 1 to 10 using do while */
#include <stdio.h>
#include<conio.h>
Void main( )
{
int i = 1 ;
clrscr( );
do
{
printf ("%d\t", i) ;
i++ ;
}
while (i<=10) ;
getch( );
}
```

**Difference between while loop and do-while loop**

| WHILE | DO-WHILE |
|---|---|
| Condition is checked first then statement(s) is executed. | Statement(s) is executed atleast once, thereafter condition is checked. |
| It might occur statement(s) is executed zero times, If condition is false. | At least once the statement(s) is executed. |
| No semicolon at the end of while.<br>while(condition) | Semicolon at the end of while.<br>while(condition); |
| If there is a single statement, brackets are not required. | Brackets are always required. |
| Variable in condition is initialized before the execution of loop. | variable may be initialized before or within the loop. |
| while loop is entry controlled loop. | do-while loop is exit controlled loop. |
| while(condition)<br>{ statement(s); } | do { statement(s); }<br>while(condition); |

**For Loop**

For loops is useful to execute a statement for a number of times. When the number of repetitions is known in advance, the use of this loop will be more efficient. Thus, this loop is also known as determinate or definite loop. The general syntax is:

```
for (counter initialization ; test condition ; increment or decrement)
{
   body of loop
}
```

```
/* Program to calculate the factorial of a number */
#include<stdio.h>
#include<conio.h>
 Void main( )
        {
                int num, i ;
                long fact = 1;
                clrscr( );
                printf ("\nEnter a number whose factorial is to be calculated:") ;
                scanf ("%d", &num) ;
                for (i=1 ; i<=num ; i++)
                fact *= i ;                          //fact = fact*i
                printf ("The factorial is : %ld", fact ) ;
                getch( );
        }
```

**Nested loop**

C programming allows using one loop inside another loop. C supports nesting of loops **Nesting of loops** is the feature in C that allows the looping of statements inside another loop.

**Syntax**

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

```
for ( init; condition; incre/decre )    //outer loop
{
  for ( init; condition; incre/decre )   //inner loop
 {
    statement(s);
  }
```

```
  statement(s);
}
```

The syntax for a **nested while loop** statement in C programming language is as follows:

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

The syntax for a **nested do...while loop** statement in C programming language is as follows:

```
do
{
  statement(s);

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

**/* WAP to print following pattern*/**

```
*
* *
* * *
* * * *
* * * * *
```

```
#include <stdio.h>
#include <conio.h>
void main()
 {
   int i, j, rows=5;
   for (i = 1; i <= rows; i++)
{
      for (j = 1; j <= i; j++)
{
         printf("* ");
      }
      printf("\n");
   }
   getch();
}
```

**/*WAP to print following pattern*/**

```
* * * * *
* * * *
* * *
* *
*
```

```
#include <stdio.h>
#include <conio.h>
void main()
 {
   int i, j, rows=5;
   for (i = rows; i >= 1; i--)
{
      for (j = 1; j <= i; j++)
{
         printf("* ");
      }
      printf("\n");
   }
   getch();
}
```

**/* WAP to print the following*/**

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
#include <stdio.h>
#include <conio.h>
void main()
```

```
 {
   int i, j, rows=5;
   for (i = 1; i <= rows; i++)
{
      for (j = 1; j <= i; j++)
{
         printf("%d ", j);
      }
      printf("\n");
   }
   getch();
}
```

**/*WAP to print the following*/**

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

#include<stdio.h>
#include <conio.h>
Void main()
{
int i, j, rows=4, number = 1;
for (i = 1; i <= rows; i++)
{
for (j = 1; j <= i; ++j)
 {
printf("%d ", number);
number++;
     }
printf("\n");
  }
getch();
```

```
}
```

**Exiting from a loop (break, continue, goto)**

**Break Statement**

The break statement terminates the loop (for, while and do...while loop) immediately when it is encountered. The break statement is used with decision making statement such as if...else.

**The break statement in C programming has the following two usages**:

- When a **break** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- It can be used to terminate a case in the **switch** statement.

If you are using nested loops, the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

**Syntax:**

The syntax for a **break** statement in C is as follows:

break;

**Example:**

```
#include<stdio.h>
#include <conio.h>
void main ()
{
int a =10;
while( a <20)
{
      printf("value of a: %d\n", a);
      a++;
if( a >15)
```

```
{
break;
}
}

getch();
}
```

## Continue Statement

The continue statement skips some statements inside the loop. The continue statement is used with decision making statement such as if...else.

For the **for** loop, **continue** statement causes the conditional test and increment portions of the loop to execute. For the **while** and **do...while** loops, **continue** statement causes the program control to pass to the conditional tests.

**Syntax:**

The syntax for a **continue** statement in C is as follows:

continue;

**Example:**

```
#include<stdio.h>
#include <conio.h>
void main()
{
int j;
for(j=0; j<=8; j++)
{
if(j==4)
{
        /* The continue statement is encountered when
           * the value of j is equal to 4.
           */
```

```
            continue;
}
/* This print statement would not execute for the
            * loop iteration where j ==4  because in that case
            * this statement would be skipped.
            */
     printf("%d ", j);
}
getch();
}
```

**goto statement**

The goto statement is known as jump statement in C. As the name suggests, goto is used to transfer the program control to a predefined label. The goto statement can be used to repeat some part of the code for a particular condition. It can also be used to break the multiple loops which can't be done by using a single break statement.

**Syntax:**

label:

**goto** label;

**Example:**

```
#include <stdio.h>
#include <conio.h>
void main()
{
  int num,i=1;
  printf("Enter the number whose table you want to print?");
  scanf("%d",&num);
  table:
```

```
  printf("%d x %d = %d\n",num,i,num*i);
  i++;
  if(i<=10)
  goto table;
  getch();
}
```

## Unit 6: Arrays and Strings

An array is a collection of same type of data item which are stored in consecutive memory locations under a common name. Suppose, we have 20 numbers of type integer and we have to sort them in ascending or descending order. If we have no array, we have to define 20 different variables like a1, a2... a20 of type int to store these twenty numbers which will be possible but inefficient, if the number of integers increase the number of variables will also be increased and defining different variables for different numbers will be impossible and inefficient. In such situation where we have multiple data items of same type to be stored, we can use array. In array system, an array represents multiple data items but they share same name. The individual elements are characterized by array name followed by one or more subscripts or indices enclosed in square brackets. The individual data items can be characters, integers, floating point numbers, etc. However, they must all be of the same type and the same storage class.

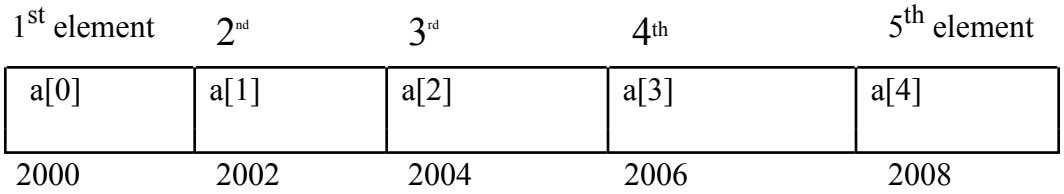
**Types of Arrays**

**Arrays can be categorized based on the number of dimension.**

- One dimensional array
- Multi dimensional array

**One or single dimensional array:**

Array with single dimension is called one-dimensional array.. In one dimensional array, there is a single subscript or index whose value refers to the individual array element which ranges from 0 to n-1 where n is the size of the array. For e.g. int a[5] ; is a declaration of one dimensional array of type int. Its elements can be illustrated as



The elements of an integer array a[5] are stored continuous memory locations. It is assumed that the starting memory location is 2000. As each integer element requires 2 bytes, subsequent element appears after gap of 2 locations. The general syntax of array is [i.e. declaration of an array :]

data_type array_name[size];

Where,

- data_type is the data type of array. It may be int, float, char, etc.
- array_name is name of the array. It is user defined name for array. The name of array may be any valid identifier.
- Size of the array is the number of elements in the array. The size is mentioned within [ ]. The size must be an int constant like 10, 15, 100, etc.

Example:

int num[5] ; i.e. num is an integer array of size 5 and store 5 integer values

char name[10] ; i.e. name is a char array of size 10 and it can store 10 characters.

float salary[50] ; i.e. salary is a float type array of size 50 and it can store 50 fractional numbers.

**Initialization of array:**

The array is initialized like follow if we need time of declaration
data_type array_name[size] = {value1, value2, ....., valuen} ;

For eg.

1. int subject[5] = {85, 96, 40, 80, 75} ;
2. char sex[2] = {'M', 'F'} ;
3. float marks[3] = {80.5, 7.0, 50.8} ;
4. int element[5] = {4, 5, 6}

In example(4), elements are five but we are assigning only three values. In this case the value to each element is assigned like following

element[0] = 4

element[1] = 5

element[2] = 6

element[3] = 0

element[4] = 0

i.e. missing element will be set to zero

**Accessing elements of array:**

If we have created an array, the next thing is how we can access (read or write) the individual elements of an array. The accessing function for array is

array_name[index or subscript]

For e.g.

```
int a[5], b[5] ;
a[0] = 30 ; /* acceptable */
b = a ; /* not acceptable */
if (a<b)
{_ _ _ _ } /* not acceptable */
```

we can assign integer values to these individual element directly:

```
a[0] = 30 ;
a[1] = 31 ;
a[2] = 32 ;
a[3] = 33 ;
a[4] = 34 ;
```

A loop can be used to input and output of the elements of array.

**Examples of single dimensional array:**

**/* Program that reads 10 integers from keyboard and displays entered numbers in the screen*/**

```
#include<stdio.h>
#include<cono.h>
Void main( )
{
        int a[10], i ;
        clrscr( ) ;
        printf (“Enter 10 numbers : \t”) ;
        for (i=0 ; i<10 ; i++)
        scanf(“%d”, &a[i]) ; /*array input */
        printf (“\n we have entered these 10 numbers : \n”) ;
         for (i=0 ; i<10 ; i++)
        printf (“\ta[%d]=%d”, i, a[i] ) ; /* array output */
        getch( ) ;
}
```

Output:

Enter 10 numbers: 10 30 45 23 45 68 90 78 34 32

We have entered these 10 numbers:

a[0] = 10 a[1] = 30 - - - - - - - - - - - - - - - - -a[9] = 32

**/*Program to find sum of marks of n students*/**

```
#include <stdio.h>
#include <conio.h>
Void main ()
{
  int n, sum = 0, c, array[20];
  printf("enter n students:");
```

```
 scanf("%d", &n);
  for (c = 0; c < n; c++)
  {
    printf("enter marks:");
    scanf("%d", &array[c]);
    sum = sum + array[c];
  }
  printf("Sum = %d\n", sum);
  getch();
}
```

**Multidimensional array**

C programming language allows programmer to create arrays of arrays known as multidimensional arrays. In multidimensional arrays, a separate pair of square brackets is required for each subscript. Thus, a two-dimensional array will require two pairs of square brackets; a three-dimensional array will require three pairs of square brackets, and so on. The two dimensional array are very useful for matrix operations.

**Declaration of multidimensional arrays**

Multidimensional arrays are declared in the same manner as one dimensional array except that a separate pair of square brackets are required for each subscript i.e. two dimensional array requires two pair of square bracket ; three dimensional array requires three pairs of square brackets, four dimensional array require four pair of square brackets, etc.

The general format for declaring multidimensional array is

```
data_type array_name [size1] [size2] ..... [sizen] ;
```

**For example:**

1. int n[5] [6] ;
2. float a[6] [7] [8] ;

**Initialization of two dimensional arrays:**

Similar to one dimensional array, two dimensional arrays can also be initialized. For e.g.

1. int [3] [2] = {1, 2, 3, 4, 5, 6) ;

   The value is assigned like follow:

   a[0] [0] = 1 ;
   a[0] [1] = 2 ;
   a[1] [0] = 3 ;
   a[1] [1] = 4 ;
   a[2] [0] = 5 ;
   a[2] [1] = 6 ;

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

   In above example, the two value in the first inner pair of braces are assigned to the array element in the first row, the values in the second pair of braces are assigned to the array element in the second row and so on. i.e.

   a[0] [0] = 1 ;  a[0] [1] = 2 ;
   a[1] [0] = 3 ;  a[1] [1] = 4 ;
   a[2] [0] = 5 ;  a[2] [1] = 6 ;

3. int a[ ] [3] = {12, 34, 23, 45, 56, 45} ; is perfectly acceptable.

It is important to remember that while initializing a 2-dimensional array, it is necessary to mention the second (column) size where as first size (row) is optional.

In above example, value assign like follow

a[0] [0] = 12  a[0] [1] = 34  a[0] [2] = 23
a[1] [0] = 45  a[1] [1] = 56  a[1] [2] = 45

int a[2] [ ] = {12, 34, 23, 45, 56, 45} ;  /* This initialization

```
        int a[ ] [ ] = {12, 34, 23, 45, 56, 45} ;          would never work*/
/* Program to read & display 2×3 matrix */
#include<stdio.h>
#include<conio.h>
void main( )
{
      int matrix [2] [3], i, j ;
      clrscr( ) ;
      for (i=0 ; i<2 ; i++)
        for (j = 0 ; j<3 ; j++)
                {
                  printf("Enter matrix [%d] [%d] : \t", i, j) ;
                  scanf("%d", &matrix [i] [j] ) ;
                }
      printf ("\n Entered matrix is : \n") ;
        for (i=0 ; i<2 ; i++)
        {
        for (j=0; j<3; j++)
          printf("%d\t", matrix [i] [j]) ;
          printf("\n") ;

        }
        getch( )
}
```

Output:

```
      Enter matrix [0] [0] :    1
      Enter matrix [0] [1] :    2
      Enter matrix [0] [2] :    3
      Enter matrix [1] [0] :    4
      Enter matrix [1] [1] :    5
```

Enter matrix [1] [2] : 6

Enter matrix is

1 2 3

4 5 6

```
/* Program to read two matrices and display their sum */
#include<stdio.h>
#include<conio.h>
Void main( )
{
    int a[3] [3], b[3] [3], s[3] [3], i, j ;
    printf("Enter first matrix : \n") ;
    for (i=0 ; i<3 ; i++)
    for (j=0 ; j<3 ; j++)
    scanf("%d", &a[i] [j]) ;
    printf("Enter second matrix : \n") ;
    for (i=0 ; i<3 ; i++)
    for (j=0 ; j<3 ; j++)
    scanf("%d", &b[i] [j]) ;
    for (i=0 ; i<3 ; i++)
    for (j=0 ; j<3 ; j++)
    s[i] [j] = a[i] [j] + b[i] [j] ;
    printf("\n The sum matrix is : \n") ;
    for (i=0 ; i<3, i++)
     {
      for(j=0 ; j<3 ; j++)
       printf("\t%d", s[i] [j]) ;
       printf("\n");  }
       getch( ) ;
      }
```

**# WAP to sorting an array element**

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

```
void main()
{
int x[]={12,32,87,90,5,4,7,2,11,9}, tmp;
int i,j;
clrscr();
printf("Array's elements before sorting are:\n");
for(i=0;i<10;i++)
printf("%d\t", x[i]);
for(i=0; i<9; i++)
for(j=i+1; j<10; j++)
if(x[i]>x[j])
{
tmp = x[i];
x[i] = x[j];
x[j] = tmp;
}
printf("Array's elements after sorting in ascending are:\n");
for(i=0;i<10;i++)
printf("%d\t", x[i]);
getch();
}
```

**String**

Strings are actually one-dimensional array of characters terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

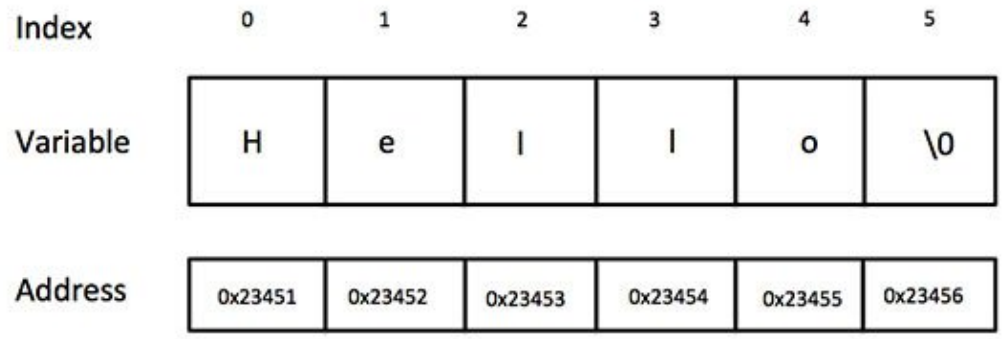
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

If you follow the rule of array initialization then you can write the above statement as follows −

char greeting[] = "Hello";

Following is the memory presentation of the above defined string in C




Actually, you do not place the **null** character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string

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

void main ()
 {

   char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
   printf("Greeting message: %s\n", greeting );
   getch();
}
```

**How to initialize strings?**

You can initialize strings in a number of ways.

```
char c[] = "abcd";

char c[50] = "abcd";

char c[] = {'a', 'b', 'c', 'd', '\0'};

char c[5] = {'a', 'b', 'c', 'd', '\0'};
```

**Example:**

```
#include <stdio.h>
#include <conio.h>
void main()
{
   char name[20];
   printf("Enter name: ");
   scanf("%s", name);
   printf("Your name is %s.", name);
getch();
}
```

**Reading a string from the keyboard:**

The easiest way to input a string from the keyboard is with the gets() library function. The general form gets() is:

gets(array_name);

To read a string, call gets() with the name of the array, with out any index, as its arguments. Upon return form gets() the array will hold the string input. The gets() function will continue to read characters until you enter a carriage return. The header file used for gets() is stdio.h

**Example:**

```
# include <stdio.h>
#include <conio.h>
Void main()
{
char str[80];
 printf ("\nEnter a string:");
gets (str);
 printf ("%s", str);
 }
```

the carriage return does not become part of the string instead a null terminator is placed at the end.

**Writing strings:**

The puts() functions writes its string argument to the screen followed by a newline. Its prototype is: puts(string);

It recognizes the same back slash code as printf(), such as "\t" for tab. As puts() can output a string of characters – It cannot output numbers or do format conversions it required faster overhead than printf(). Hence, puts() function is often used when it is important to have highly optimized code.

For example, to display "hello" on the screen:

```
puts("hello");
```

**String Handling Functions in C**

C programming language provides a set of pre-defined functions called **string handling functions** to work with string values. The string handling functions are defined in a header file called **string.h**. Whenever we want to use any string handling function we must include the header file called **string.h**.

**The following table provides most commonly used string handling function and their use:**

| Function | Syntax (or) Example | Description |
|---|---|---|
| **strcpy()** | strcpy(string1, string2) | Copies string2 value into string1 |
| **strrev()** | strrev(string1) | It reverses the value of string1 |
| **strlen()** | strlen(string1) | returns total number of characters in string1 |
| **strcat()** | strcat(string1,string2) | Appends string2 to string1 |
| **strlwr()** | strlwr(string1) | Converts all the characters of string1 to lower case |

**Example:**

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
void main ()
{
char str1[12] = "Hello";
char str2[12] = "World";
char str3[12];
   int  len ;
   /* copy str1 into str3 */
   strcpy(str3, str1);
   printf("the copy from str1 is :  %s\n", str3 );

   /* concatenates str1 and str2 */
   strcat( str1, str2);
   printf("the concatenate string is:   %s\n", str1 );
   /* total lenghth of str1 after concatenation */
```

```
  len = strlen(str1);
  printf("the total length of string is :  %d\n", len );
  /* reverse string*/
  strrev(str1);
  printf(“the reverse string is: %s\n, str1);
  getch();
}
```

# Unit-7 Functions

A function is defined as a self contained block of statements that performs a particular task. This is a logical unit composed of a number of statements grouped into a single unit. It can also be defined as a section of a program performing a specific task. [Every C program can be thought of as a collection of these functions.] Each program has one or more functions. The function main ( ) is always present in each program which is executed first and other functions are optional.

**Advantages of using Functions**

The advantages of using functions are as follows:

1. Generally a difficult problem is divided into sub problems and then solved. This divide and conquer technique is implemented in C through functions. A program can be divided into functions, each of which performs some specific task. So, the use of C functions modularizes and divides the work of a program.
2. When some specific code is to be used more than once and at different places in the program, the use of function avoids repetition of that code.
3. The program becomes easily understandable. It becomes simple to write the program and understand what work is done by each part of the program.
4. Functions can be stored in a library and reusability can be achieved.

**Types of Functions**

C program has two types of functions:

1. Library Functions
2. User defined functions

**Library Functions:**

These are the functions which are already written, compiled and placed in C Library and they are not required to be written by a programmer. The function's name, its return type, their argument number and types have been already defined. We can use these functions as required. For example: printf(), scanf(), sqrt(), getch(), etc.

## User defined Functions:

These are the functions which are defined by user at the time of writing a program. The user has choice to choose its name, return type, arguments and their types. The job of each user defined function is as defined by the user. A complex C program can be divided into a number of user defined functions.

### Function Declaration or Prototype

The function declaration or prototype is model or blueprint of the function. If functions are used before they are defined, then function declaration or prototype is necessary. Many programmers prefer a "top-down" approach in which main appears ahead of the programmer defined function definition. Function prototypes are usually written at the beginning of a program, ahead of any user-defined function including main (). Function prototypes provide the following information to the compiler.

- The name of the function
- The type of the value returned by the function
- The number and the type of arguments that must be supplied while calling the function.

In "bottom-up" approach where user-defined functions are defined ahead of main () function, there is no need of function prototypes. The general syntax of function prototype is

```
return_type function_name (type1, type2,....., typen) ;
```

where, return_type specifies the data type of the value returned by the function. A function can return value of any data type. If there is no return value, the keyword void is used. type1, type2,....., typen are type of arguments. Arguments are optional. For example:

```
int add (int, int) ; /* int add (int a, int b) ;*/

void display (int a) ; /* void display (int); */
```

**Function definition:**

A function definition is a group of statements that is executed when it is called from some point of the program. The general syntax is

return_type function_name (parameter1, parameter2, ....., parametern)

{-----

statements ;

-----

}

Where,

return_type is the data type specifier of data returned by the function.

function_name is the identifier by which it will be possible to call the function.

Parameters(as many as needed) : Each parameter consists of a data type specifier followed by an identifier like any regular variable declaration. (for eg: int x) and which

acts within the function as a regular local variable. They allow to pass arguments to the function when it is called. The different parameters are separated by commas.

statements is the function's body. It is a block of statements surrounded by braces{}.

The first line of the function definition is known as function header.

```
int addition (int a, int b);    /* function prototype */
void main( )
{
int a ;
  a = addition (5,3) ; /*function call */
  printf ("The result is%d",a) ;

}
```

```
 int addition(int a, int b) /* function header */
{
int r;
 r = a+b
 return r;
 }
```

**return statement:**

The return statement is used in a function to return a value to the calling function. It may also be used for immediate exit from the called function to the calling function without returning a value.

This statement can appear anywhere inside the body of the function. There are two ways in which it can be used:

```
return ;

return (expression) ;
```

where return is a keyword. The first form of return statement is used to terminate the function without returning any value. In this case only return keyword is written.

**Calling a function**

A function can be called by specifying its name, followed by a list of arguments enclosed in parenthesis and separated by commas in the main() function. The syntax of the call if function return_type is void is: function_name (parameter name);

If function return int, float or any other type value then we have to assign the call to same type value like

variable = function_name(parameter) ;

**Actual and Formal Parameters:**

The arguments are classified into

1. Actual
2. Formal

Actual: When a function is called some parameters are written within parenthesis. These are known as actual parameter. For example

```
main()
{-----
  convert(c) ;  actual parameter
  -----
}
```

Formal parameters are those who are written in function header. For example

```
int convert (int a)
                formal parameter
{
      }
```

**Some example programs using function**

```
# WAP to add two numbers using function
#include<stdio.h>
#include<conio.h>
int sum (int , int );
void main()
{
  int num1, num2, total;
  printf("\nEnter the two numbers:");
  scanf("%d %d", &num1, &num2);
  total=sum(num1, num2); //Call Function Sum With Two Parameters
```

```
   printf("Addition of two number is:%d",total);
   getch();
}
int sum(int num1, int num2)
 {
   int num3;
   num3 = num1 + num2;
   return (num3);
}
```

**# WAP to find largest among two input number using function**

```
#include<stdio.h>
#include <conio.h>
int max(int num1,int num2); //function declaration

void main ()
{
int a =100;
int b =200;
int largest;
largest = max(a, b);      //calling a function

printf("Max value is : %d\n", largest );

getch();
}

int max (int num1,int num2)  // function header
{

int result;

if(num1 > num2)
     result = num1;
else
     result = num2;

return result;
}
```

**# program to find the factorial using function**

```
#include <stdio.h>
#include <conio.h>
int fact(int);
 void main()
{
 int no,factorial;
 printf("Enter a number to calculate it's factorial:");
 scanf("%d",&no);
 factorial=fact(no);
 printf("Factorial of the num(%d) = %d\n",no,factorial);
 getch();
}
int fact(int n)
{
int i,f=1;
for(i=1;i<=n;i++)
   {
      f=f*i;
   }
return f;
}
```

**Passing Arguments to a Function**

Arguments can be passed in function in one of the following two ways

- Passing by value (sending the values of the arguments)
- Passing by reference/address (sending the address of the arguments)

In the first method the 'value' of each actual argument in the calling function is copied into corresponding formal argument of the called function. With this method, changes made to the formal arguments in the called function have no effect on the values of the actual arguments in the calling function.

**Example:**

```
#include <stdio.h>
#include<conio.h>
void swap(int x, int y);
void main()
{
int a =5;
int b= 10;
printf("\nBefore calling swap()");
printf("\na= %d ",a);
printf("\nb= %d",b);
swap(a,b);
printf("\nAfter calling swapv()");
printf("\na= %d ",a);
printf("\nb= %d",b);
getch();
}
void swap(int x, int y)
{
int temp;
temp=x;
x=y;
y=temp;
printf("\nInside swap()");
printf("\na= %d ",x);
```

```
printf("\nb= %d",y);
}
```

In the second method (call by reference) the address of actual arguments in the calling function are copied into formal arguments of the called function. This using the formal arguments in the called function we can make changes in the actual arguments of the calling function.

**Example:**

```
#include<stdio.h>
#include<conio.h>
void swap(int *x, int *y);
void main()
{
int a =5;
int b= 10;
printf("\nBefore calling swap()");
printf("\na= %d ",a);
printf("\nb= %d",b);
swap(&a,&b);
printf("\nAfter calling swap()");
printf("\na= %d ",a);
printf("\nb= %d",b);
getch();
}
void swap(int *x, int *y)
{
int temp;
temp= *x;
```

```
*x= *y;
*y= temp;
printf("\nInside swap()");
printf("\na= %d ",*x);
printf("\nb= %d",*y);
}
```

**Recursive Function**

- Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied.
- The process is used for repetitive computations in which each action is stated in terms of a previous result.
- Recursive functions are useful in evaluating certain types of mathematical function. In order to solve a problem recursively, two conditions must be satisfied.

  1. First, the problem must be written in a recursive form,
  2. Second, the problem statement must include a stopping condition.

- Care must be taken to define functions which will not call themselves indefinitely, otherwise your program will never finish.

```
/* WAP to find the factorial of a number using recursive method */
#include<stdio.h>
#include<conio.h>
long int factorial (int n)
{
    if (n = = 1)
      return (1) ;
    else
      return (n*factorial(n-1)) ;       // function calls itself
}
```

```
Void main( )
{
int num ;
    printf (“Enter a number : ”) ;
    scanf (“%d”, &num) ;
    printf (“The factorial is %ld”, factorial (num)) ;
}
```

**Passing Array to Function**

In C, there are various general problems which require passing more than one variable of the same type to a function. For example, consider a function which sorts the 10 elements in ascending order. Such a function requires 10 numbers to be passed as the actual parameters from the main function. Here, instead of declaring 10 different numbers and then passing into the function, we can declare and initialize an array and pass that into the function. This will resolve all the complexity since the function will now work for any number of values.

As we know that the array_name contains the address of the first element. Here, we must notice that we need to pass only the name of the array in the function which is intended to accept an array. The array defined as the formal parameter will automatically refer to the array specified by the array name defined as an actual parameter.

**Consider the following syntax to pass an array to the function.**

Function name (array name);  //passing array

**Methods to declare a function that receives an array as an argument**

There are 3 ways to declare the function which is intended to receive an array as an argument.

**First way:**

return_type function(type arrayname[])

Declaring blank subscript notation [] is the widely used technique.

**Second way:**

return_type function(type arrayname[SIZE])

Optionally, we can define size in subscript notation [].

**Third way:**

return_type function(type *arrayname)

You can also use the concept of a pointer.

**Example:**

```
#include<stdio.h>
#include<conio.h>
double getAverage(int arr[], int size);
void main ()
{
  int balance[5] = {1000, 2, 3, 17, 50};
  double avg;
  avg = getAverage( balance, 5 ) ;
  printf( "Average value is: %lf ", avg );
  getch();
}
double getAverage(int arr[], int size) {
  int i;
  double avg;
  double sum = 0;
  for (i = 0; i < size; i++)
{
     sum += arr[i];
  }
  avg = sum / size;
  return avg;
}
```

## C - Scope Rules

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. There are three places where variables can be declared in C programming language −

- Inside a function or a block which is called **local variables.**
- Outside of all functions which is called **global variables**.
- In the definition of function parameters which are called formal parameters.

Let us understand what local and global variables are.

### Local Variables

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. The following example shows how local variables are used. Here all the variables a, b, and c are local to main() function.

```
#include <stdio.h>
void main ()
{
  /* local variable declaration */
  int a, b;
  int c;
  /* actual initialization */
  a = 10;
  b = 20;
  c = a + b;
  printf ("value of a = %d, b = %d and c = %d\n", a, b, c);
}
```

**Global Variables**

Global variables are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. The following program show how global variables are used in a program.

```
#include <stdio.h>

/* global variable declaration */

int g;

Void main ()

{

  /* local variable declaration */

  int a, b;

  /* actual initialization */

  a = 10;

  b = 20;

  g = a + b;

printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

 }
```

A program can have same name for local and global variables but the value of local variable inside a function will take preference. Here is an example

```
#include <stdio.h>

/* global variable declaration */
```

```
int g = 20;

Void main ()

{

  /* local variable declaration */

  int g = 10;

  printf ("value of g = %d\n",  g);

 }
```

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

```
Value of g = 10
```

## Unit-8 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 */
```

**Indirection & Address Operators**

- To manipulate data using pointers, the C language provides two operators: *address (&)* and *indirection (**)*. These are unary prefix operators.
- The address-of operator (**&**) gives the address of its operand. The result of the address operation is a pointer to the operand. The type addressed by the pointer is the type of the operand.

**Declaring and assigning pointer**

Pointer variable must be declared as pointer before we use them. The declaration of a pointer variable takes the following form:

***data type* *ptname;**

The following examples use these declarations:

int *pa,

int x;

int a[20];

This statement uses the address-of operator:

pa = &a[5];

pa = &x;

The indirection operator (*) is used in this example to access the int value at the address stored in pa.

**Example:**

```
#include<stdio.h>
#include <conio.h>
Void 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: %u\n",&var);        /* address stored in pointer variable */
  printf("Address stored in ip variable: %u\n", ip );   /* access the value using the pointer */
  printf("Value of *ip variable: %d\n",*ip );
  getch();
}
```

**Pointer Arithmetic Operations**

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 -

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 2 bytes next to the current location in 16 bit system. 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.

**Pointer Increments and Scale Factor**

•characters1 byte

•integers2 byte

•floats4 bytes

•long integers4 bytes

•doubles8 bytes

**Pointer to Pointer (Double pointer)**

A pointer to a pointer is a form of multiple indirections, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.


A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, the following declaration declares a pointer to a pointer of type int

int **var ;

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice.

**Example:**

```
#include<stdio.h>
#include <conio.h>
void main ()
{
int var;
int *ptr;
int **pptr;
var=3000;

/* take the address of var */
  ptr =&var;

/* take the address of ptr using address of operator & */
  pptr =&ptr;

/* take the value using pptr */
  printf("Value of var = %d\n",var);
  printf("Value of first pointer = %d\n",*ptr );
  printf("Value of second pointer = %d\n",**pptr);

getch();
}
```

**Pointer to an Array**

An array name is a constant pointer to the first element of the array. Therefore, in thedeclaration: 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 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.

Once you store the address of first element in p, you can access array elements using *p, *p + 1, * p + 2 and so on.

**Example:**

```
#include <stdio.h>
#include<conio.h>
void 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) );
  }
  getch();
}
```

**Passing pointers to functions:**

A pointer can be passed to a function as an argument. Passing a pointer means passing address of a variable instead of value of the variable. As address is passed in this case, this mechanism is also k\a call by address or call by reference. When pointer is passed to a function, while function calling, the formal argument of the function must be compatible with the passing pointer i.e. if integer pointer is being passed, the formal argument in function must be pointer of the type integer and so on. As address of variable is passed in this mechanism, if value in the passed address is changed within function, the value of actual variable also changed.

```
/* Program to illustrate the use of passing pointer to a function */
#include<stdio.h>
#include<conio.h>
 void addGraceMarks (int *m)
        {
        *m = *m+10;
        }
  void main( )
        {
          int marks;
          printf(“Enter actual marks: ”);
          scanf(“%d”,&marks);
          add GraceMarks(&marks);          \* Passing address *\
           printf(“\n The graced marks is: %d”, marks);
        }
```

## Unit 9: Structure and Union

**Structure**

Structure is a collection of data item. The data item can be different type, some can be int, some can be float, and some can be char and so on. The data item of structure is called member of the structure.

In other words we can say that heterogeneous data types can be grouped to form a structure. In some languages structure is known as record. The different between array and structure is the element of an array has the same type while the element of structure can be of different type. Another different is that each element of an array is referred to by its position while each element of structure has a unique name.

The **struct** keyword is used to define the structure. Let's see the syntax to define structure in C.

```
struct structure_name
{
   data_type member1;
   data_type member2;
   .
   .
   data_type memeberN;
};
```

Let's see the example to define a structure for an entity employee in c.

```
struct employee
{
   int id;
   char name[20];
   float salary;
};
```

**Declaring structure variable**

We can declare a variable for the structure so that we can access the member of the structure easily. There are two ways to declare structure variable:

1. By struct keyword within main() function
2. By declaring a variable at the time of defining the structure.

**1st way:**

Let's see the example to declare the structure variable by struct keyword. It should be declared within the main function.

```
struct employee
{   int id;
   char name[50];
   float salary;
};
```

Now write given code inside the main() function.

```
struct employee e1, e2;
```

The variables e1 and e2 can be used to access the values stored in the structure.

**2nd way:**

Let's see another way to declare variable at the time of defining the structure.

```
struct employee
{
   int id;
   char name[50];
   float salary;
}e1,e2;
```

**Accessing members of the structure**

There are two ways to access structure members:

1. By **.** (member or dot operator)
2. By -> (structure pointer operator)

Let's see the code to access the *id* member of *p1* variable by**.** (member) operator.

p1.id

/* Create a structure named student that has name, roll, marks and remarks as members. Assume appropriate types and size of member. WAP using structure to read and display the data entered by the user */

```
# include <stdio.h>
# include <conio.h>
struct student
{
char name[20];
 int roll;
float marks;
char remark;
} s;
Void main()
{
//struct student s;
 clrscr( );
printf("enter name: \t");
gets(s.name);
printf(" \nenter roll:\t");
scanf("%d", &s.roll);
printf("\n enter marks: \t");
scanf("%f", &s.marks);
```

```
printf("enter remarks p for pass or f for fail: \t")
s.remark = getch( )
printf("The student's information is \n");
printf("Student Name \t Roll \t Marks \t Remarks");
printf("\n - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -\n");
printf("%s\t%d\t%.2f\t%c", s.name, s.roll, s.marks, s.remark);
getch( );
}
```

**Array of Structures**

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.

For example:

```
struct employee
  {
  char name[20];
   int empid;
  float salary;
  } emp[10];
```

Where emp is an array of 10 employee structures. Each element of the array emp will contain the structure of the type employee. The another way to declare structure is

```
struct employee
{
    char name[20];
    int empid;
```

```
    float salary;
};
struct employee emp[10];
```

/* Create a structure named student that has name, roll, marks and remarks as members. Assume appropriate types and size of member. Use this structure to read and display records of 5 student */

```
#include<stdio.h>
#include<conio.h>
struct student
{
Char name[30];
int roll;
float marks;
char remark;
};
Void main()
{
struct student st[5];
    int i;
    clrscr( );
    for(i=0; i<5; i++)
{
    printf("\n Enter Information of student No%d\n"; i+1);
    printf("Name: \t");
    scanf("%s", s[i].name);
    printf("\n Roll: \t");
    scanf("%d", &s[i].roll);
    printf("\n Marks:\t");
```

```
    scanf("%f", &s[i].marks);
    printf("remark(p/f): \t");
    s[i].remark = getch( );
  }
    printf("\n\n The Detail Information is \n");
    printf("Student Name: \t Roll \t Marks \t Remarks")
    printf("\n_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ \n")

    for(i=0; i<5; i++)
    printf("%s\t\t%d\t%.2f \t%c\n".s[i].name, s[i].roll, s[i].marks, s[i].remark);
    getch( );
  }
```

**Nested Structure (Structure within structure)**

Sometimes the member of structure needs to have multiple values, for the case the members should be another variable of another structure type. C allows the member structure to be the structure. The mechanism is called nesting of the structure. Nesting enables to build powerful data structures.

Nested structure in C is nothing but structure within structure. One structure can be declared inside other structure as we declare structure members inside a structure.

/* Create a structure named date that has day, month and year as its members. Include this structure as a member in another structure named employee which has name, id, salary, as other members. Use this structure to read and display employee's name, id, DOB and salary *|

```
#include<stdio.h>
#include<conio.h>
 void main( )
      {
        struct date
        {       int day;
```

```
            int month;
            int year;
  };
  struct employee
  {
  char name[20];
  int id;
  struct date dob;
  float salary;
} emp;
printf("Name of Employee: \t");
scanf("%s", emp.name);
printf("\n ID of employee: \t");
scanf("%d", & emp.id);
printf("\n Day of Birthday: \t");
scanf("%d", &emp.dob.day);
printf("\n month of Birthday: \t");
scanf("%d", &emp.dob.month);
printf("\n Year of Birthday: \t");
scanf("%d", &emp.dob.year);
printf("salary of Employee: \t");
scanf("%d", &emp.salary);
printf("\n\n The Detail Information of Employee");
printf("\n Name \t id \t day \t month \t year \t salary");
printf("\n - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - n");
printf("%s\t%d\t%d\t%d\t%d\t%.2f",emp.name,emp.id,emp.dob.day, emp.dob.month,
  emp.dob.year, emp.salary);

}
```

**Structures and pointers:**

Pointers can be used also with structure. To store address of a structure type variable, we can define a structure type pointer variable as normal way. Let us consider a structure book that has members name, page and price. It can be declared as

```
struct book
{
char name[20];
int page;
float price;
};
```

Then we can define structure variable and pointer variable of structure type

```
struct book b;          /*  b is structure variable */
Struct book *p;         /*  p is pointer variable of structure type */
```

Where b is simple variable of structure type book where as p is pointer type variable which points or can store address of structure book type variable.

This declaration for a pointer to structure does not allocate any memory for a structure but allocates only for a pointer. To use structure's members through pointer p, memory must be allocated for a structure by using function malloc( ) or by adding declaration and assignment as given below

```
p = &b;
```

Here, the base address of b can assign to p pointer.

An individual structure member can be accessed in terms of its corresponding pointer variable by writing

```
ptr_variable -> member
```

where -> is called arrow operator and there must be pointer to the structure on the left side of this operator.

Now, the members name, pages, and price of book can be accessed as

```
b.name or p ->name or (*p).name
b.pages or p->pages or (*p).pages
```

b.price or p->price or (*p).price

```
#include<stdio.h>
#include<conio.h>
 void main( )
        {
          struct book;
          {
          char name[20];
          int pages;
          float price;
        };
        struct book b, *p;
        clrscr( );
        printf("Enter Book's Name: \t");
        gets(b.name);
        printf("\n Number of pages: \t");
        scanf("%d", &b.pages);
        printf("\n price of the book:\t");
        scanf("%f", &b.price),
        p = &b;
        printf("\n \n Book Information Using with arrow operator");
        printf("\n Book Name=%s\t Pages=%d\t price=%.2f, p->name, p-> pages, p->price);
        getch( );

        }
```

**Passing structure to function**

- A structure can be passed to any function from main function or from any sub function.
- Structure definition will be available within the function only.

- It won't be available to other functions unless it is passed to those functions by value or by address (reference).
- Else, we have to declare structure variable as global variable. That means, structure variable should be declared outside the main function. So, this structure will be visible to all the functions in a C program.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
struct student
{
        int id;
        char name[20];
        float percentage;
};
void func (struct student record);
void main()
{
        struct student record;
        record.id=1;
        strcpy (record.name, "Raju");
        record.percentage = 86.5;
        func(record);
}
void func (struct student record)
{
        printf(" Id is: %d \n", record.id);
        printf(" Name is: %s \n", record.name);
printf(" Percentage is: %f \n", record.percentage);
getch();
}
```

**Unions**

- A union is a collection of variables of different types, just like a structure.
- However, with unions, we can only store information in one field at any one time. You can picture a union as like a chunk of memory that is used to store variables of different types.
- Once a new value is assigned to a field, the existing data is wiped over with the new data.
- The major distinction between structure and union is in terms of storage.
- In structures, each member has its own storage location, whereas all the members of a union use the same location.
- This implies that, although a union may contain many members of different types, it can handle only one member at a time.

Declaration of the union is the same as structure. Instead of the keyword struct the keyword union is used.

```
union union name
 {
 datatype member1 ;
  datatype member2 ;
. . . . . .
 datatype memberN ;
}
```

**Example:**

```
union item
{
int m;
float x;
```

```
char c;
}code;
```

**Accessing Union Fields**

To access the fields of a union, use the dot operator (.) just as you would for a structure. When a value is assigned to one member, the other member(s) get whipped out since they share the same memory. Using the example above, the precise time can be accessed like this:

- code.m
- code.x
- code.c

**Example:**

```
#include <stdio.h>
#include<conio.h>
#include <string.h>
union Data
{
  int i;
  float f;
  char str[20];
}data;
void main( )
{
  data.i = 10;
  data.f = 220.5;
  strcpy( data.str, "C Programming");
  printf( "data.i : %d\n", data.i);
  printf( "data.f : %f\n", data.f);
  printf( "data.str : %s\n", data.str);
  getch();
}
```

## Unit 10: File Handling in C

A file is a place on the disk where a group of related data is stored. The data file allows us to store information permanently and to access and alter that information whenever necessary. Programming language C has various library functions for creating and processing data files. Mainly, there are two types of data files:

1. High level (standard or stream oriented) files.
2. Low level (system oriented) files.

In high level data files, the available library functions do their own buffer management where as the programmer should do it explicitly in case of lower level files.

The standard data files are again subdivided into text files and binary files. The text files consist of consecutive characters and these characters can be interpreted as individual data item. The binary files organize data into blocks containing contiguous bytes of information. For each, binary and text files, there are a number of formatted and unformatted library functions in C.

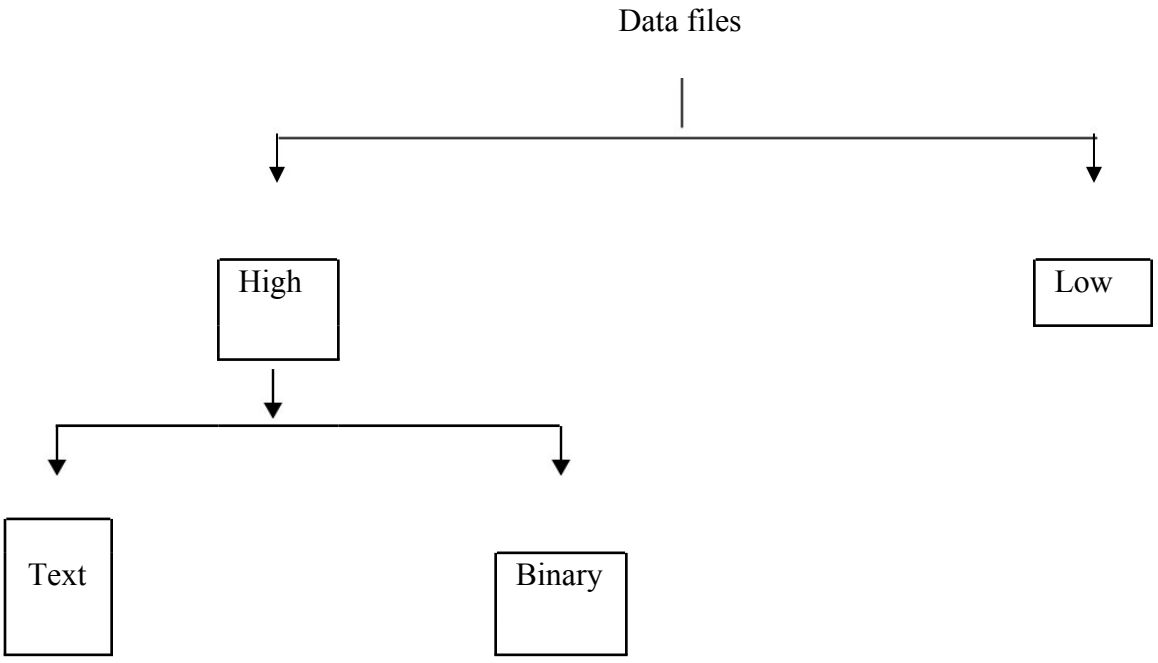



Fig: classification of files

**Stream**

- A stream is a series of ordered bytes.
- A stream is a source or destination of data that may be associated with a disk or other peripheral.
- The c library stdio.h supports text streams and binary streams. A text stream is a sequence of lines; each line has zero or more characters and is terminated by '\n'.
- A binary stream is a sequence of unprocessed bytes that record internal data, with the property that if it is written, then read back on the same system, it will compare equal.
- A stream is connected to a file or device by opening it; the connection is broken by closing the stream.
- Opening a file returns a pointer to an object of type FILE, which records whatever information is necessary to control the stream. We will use “file pointer” and “stream” interchangeably when there is no ambiguity.

**Working with file**

- C supports a number of functions that have the ability to perform basic file operations, which include:
  1-Naming a file
  2-Opening a file
  3-Reading from a file
  4-Writing data into a file
  5-Closing a file
- When working with a stream-oriented data file, the first step is to establish the buffer area, where information is temporarily stored while being transferred between the computer’s memory and the data file.
- The buffer area is established by writing
  FILE*ptvar;

**Opening File**

- The library function fopen is used to open a file. This function is typically written as
- ptvar= fopen(file-name, file-mode);
- Where file-name and file-mode are strings that represent the name of the data file and the manner in which the data file will be utilized and ptvar is pointer to the FILE structure.
- The FILE structure contains information about the file being used, such as its current size, its location in memory etc.

**Closing File**

- When we have finished reading from the file, we need to close it.
- This is done using the function fclose() through the statement,
- fclose(fp);

**File Opening Modes**

A file can be opened in different modes. Below are some of the most commonly used modes for opening or creating a file.

- r : opens a text file in reading mode.
- w : opens or creates a text file in writing mode.
- a : opens a text file in append mode.
- r+ : opens a text file in both reading and writing mode. The file must exist.
- w+ : opens a text file in both reading and writing mode. If the file exists, it's truncated first before overwriting. Any old data will be lost. If the file doesn't exist, a new file will be created.
- a+ : opens a text file in both reading and appending mode. New data is appended at the end of the file and does not overwrite the existing content

**Library functions for reading/writing from/to a file:**

**1. Unformatted I/O functions:**

a) Character I/O functions:

fgetc( ): It is used to read a character from a file. Its syntax is

char_variable = fgetc(file_ptr_variable);

fputc( ): It is used to write a character to a file. Its syntax is

fputc(char_variable, file_ptr, vriable);

b) String I/O functions:

fgets( ): It is used to read string from file. Its syntax is

fgets(string, int_value, file_ptr_variable);

fputs( ): It is used to write a string to a file. Its syntax is

fputs(string, file_ptr_variable);

**2. Formatted I/O functions:**

fprintf( ): This function is used to write some integer, float, char or string to a file. Its syntax is:

fprintf(file_ptr_variable,"control string", list variables);

fscanf( ); This function is used to read some integer, float char or string from a file. Its syntax is

fscanf(file_ptr_variable,"control string", & list_variables);

**Example of writing to a text file:**

```
#include<stdio.h>

#include<conio.h>

void main()

{

FILE *p;

char ch;
```

```
p = fopen("hello.txt", "w");
printf(“enter a character:”);
scanf("%c", &ch);
fputc(ch,p);
fclose(p);
getch();
}
```

**Example of reading from a text file:**

```
#include<stdio.h>
#include<conio.h>
void main()
{
FILE *fp;
char ch;
fp = fopen("hello.txt", "r");
ch=getc(fp);
printf("Character read from file is:%c",ch)
fclose(fp);
getch();
}
```

**Program that writes strings to a file**

```
#include<stdio.h>
#include<string.h>
void main()
{
FILE *fp;
char s[80];
fp=fopen("F1.txt","w");
if(fp==NULL)
{
puts("Cannot open target file");
fclose(fp);
exit(1);
}
printf("\nEnter a few lines of text:\n");
while(strlen(gets(s))>0)
{
fputs(s,fp);
}
fclose(fp);
getch();
}
```

**Program that reads strings from a file**

```
#include<stdio.h>
#include<conio.h>
void main()
{
FILE *fp;
char s[80];
fp=fopen("F1.txt","r");
if(fp==NULL)
{
puts("Cannot open target file");
fclose(fp);
exit(1);
}
while(fgets(s,79,fp)!=NULL)
{
printf("%s\n",s);
}
fclose(fp);
getch();
}
```

**/* Program to understand fprintf( ) */**

```
#include<stdio.h>
 #include<conio.h>
 void main()
  {
  FILE *fp;
  int roll;
  char name[25];
  float marks;
  fp = fopen("file.txt","w");
  if(fp == NULL)
  {
   printf("\nCan't open file or file doesn't exist.");
   exit(0);
   }
   printf("enter information:");
    {
     printf("\nEnter Roll : ");
     scanf("%d",&roll);
     printf("\nEnter Name : ");
     scanf("%s",name);
     printf("\nEnter Marks : ");
     scanf("%f",&marks);
     fprintf(fp,"%d%s%f",roll,name,marks);
}
    fclose(fp);
    getch(); }
```

**/* Program to understand fscanf( ) */**

```
#include<stdio.h>
#include<conio.h>
void main()
{
FILE *fp;
int roll;
char name[20];
float marks;
fp = fopen("file.txt","r");
if(fp == NULL)
{
 printf("\nCan't open file or file doesn't exist.");
 exit(0);
 }
 printf("\nData in file...\n");
 while((fscanf(fp,"%d%s%f",&roll,name,&marks))!=EOF)
 printf("%d%s%f",roll,name,marks);
 fclose(fp);
 getch();
}
```

**Random access in File**

Random accessing of files in C language can be done with the help of the following functions −

- ftell ( )
- rewind ( )
- fseek ( )

**ftell ( )**

It returns the current position of the file ptr.

The syntax is as follows −

```
int n = ftell (file pointer)
```

For example,

```
FILE *fp;
int n;
_____
_____
_____
n = ftell (fp);
```

**Note** − ftell ( ) is used for counting the number of characters which are entered into a file.

**rewind ( )**

It makes file ptr move to beginning of the file.

The syntax is as follows −

```
rewind (file pointer);
```

For example,

```
FILE *fp;
  -----
  -----
  rewind (fp);
  n = ftell (fp);
  printf ("%d", n);
```

**Output**

The output is as follows −

0 (always).

**fseek ( )**

It is to make the file pntr point to a particular location in a file.

The syntax is as follows −

```
fseek(file pointer, offset, position);
```

## Unit 11 C Preprocessor

The C Preprocessor is not part of the compiler, but is a separate step in the compilation process. In simplistic terms, a C Preprocessor is just a text substitution tool and they instruct compiler to do required pre-processing before actual compilation. We'll refer to the C Preprocessor as the CPP. All preprocessor commands begin with a pound symbol #. It must be the first non blank character, and for readability, a preprocessor directive should begin in first column. Following section lists down all important preprocessor directives:

| Directive | Description |
|---|---|
| #define | Substitutes a preprocessor macro |
| #include | Inserts a particular header from another file |
| #undef | Undefines a preprocessor macro |
| #ifdef | Returns true if this macro is defined |
| #ifndef | Returns true if this macro is not defined |
| #if | Tests if a compile time condition is true |
| #else | The alternative for #if |
| #elif | #else an #if in one statement |
| #endif | Ends preprocessor conditional |
| #error | Prints error message on stderr |
| #pragma | 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.

**#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**

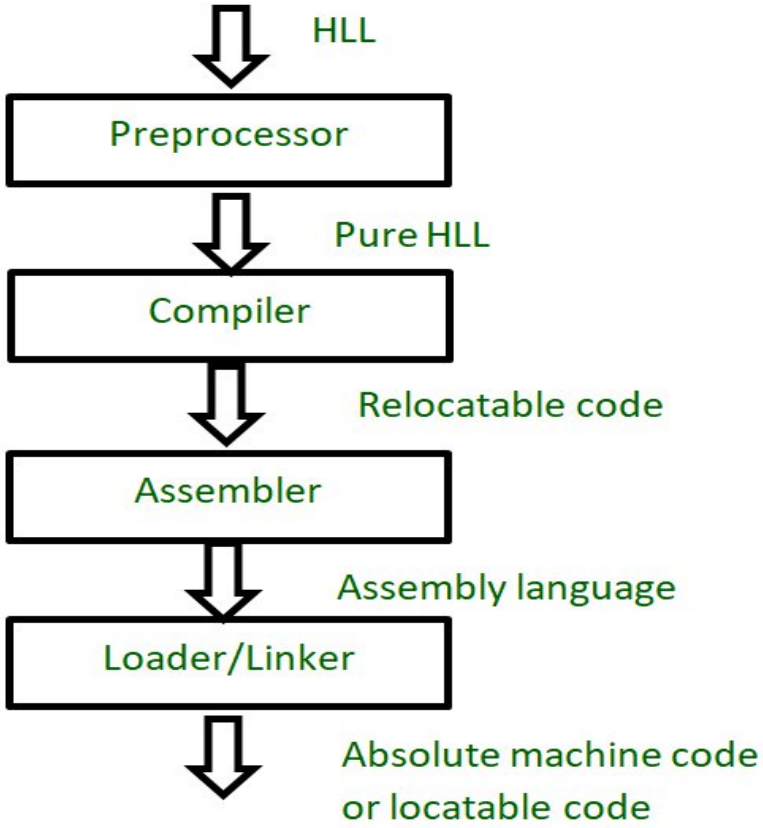
#define FILE_SIZE 42

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

**How the preprocessor works**

A Preprocessor is a system software (a computer program that is designed to run on computer's hardware and application programs). It performs preprocessing of the High Level Language (HLL). Preprocessing is the first step of the language processing system. Language processing system translates the high level language to machine level language or absolute machine code (i.e. to the form that can be understood by machine).

- The preprocessor doesn't know about the scope rules of C. Preprocessor directives like #define come into effect as soon as they are seen and remain in effect until the end of the file that contains them; the program's block structure is irrelevant.



**A Preprocessor mainly performs three tasks on the HLL code:**

**Removing comments:** It removes all the comments. A comment is written only for the humans to understand the code. So, it is obvious that they are of no use to a machine. So, preprocessor removes all of them as they are not required in the execution and won't be executed as well.

**File inclusion:** Including all the files from library that our program needs. In HLL we write **#include** which is a directive for the preprocessor that tells it to include the contents of the library file specified. For example, #include<stdio.h> will tell the preprocessor to include all the contents in the library file stdio.h.

**Macro expansion:** Macros can be called as small functions that are not as overhead to process. If we have to write a function (having a small definition) that needs to be called recursively (again and again), then we should prefer a macro over a function.

## Macro

A macro is a segment of code which is replaced by the value of macro. Macro is defined by #define directive. There are two types of macros:

1. Object-like Macros
2. Function-like Macros

### Object-like Macros

The object-like macro is an identifier that is replaced by value. It is widely used to represent numeric constants. For example:

#define PI 3.14

Here, PI is the macro name which will be replaced by the value 3.14.

### Function-like Macros

The function-like macro looks like function call. For example:

#define MIN(a,b) ((a)<(b)?(a):(b))

Here, MIN is the macro name.

**#if and #endif directives**

The #if preprocessor directive takes condition in parenthesis, if condition is true, then the statements given between #if and #else will get execute. If condition is false, then statements given between #else and #endif will get execute.

**Syntax of #if-#else-#endif Preprocessor Directive**

```
#if(condition)
    - - - - - - - - - -
    - - - - - - - - - -
#else
    - - - - - - - - - -
    - - - - - - - - - -
#endif
```

**Example of #if-#else-#endif Preprocessor Directive**

```
#include<stdio.h>
#include<conio.h>
#define MAX 45
void main()
{
```

```
        #if MAX > 40
            printf("Yes, MAX is greater then 40.");
        #else
            printf("No, MAX is not greater then 40.");
        #endif
        getch();
    }
```