C Programming Language

Introduction to C: -

C is a high level language. It is both a general purpose and a specific purpose programming language. C is the most popular and common programming language for every application developer. It was developed at the Bell Telephone Laboratory, USA (Now AT & T), in 1972. It was developed by Dennis Ritchie and Brian Kernighan.

C is derived from two early programming languages such as BCPL (Basic Combined Programming Language) and B language. Dennis Ritchie developed a new version of B and named it as C. He selected the name C for his new language because C comes after B in alphabetical order which indicates advancement to B.

Characteristics of C:-

1. C is a general purpose and stuctured programming language.
2. It helps in development of system software.
3. It has a rich set of operators and no rigid format.
4. Ability to extend itself by adding functions to its library.

Applications of C:-

Because of its portability and efficiency, C is used to develop the system as well as application software. Some of the system and application software listed below are:

1. Operating Systems
2. Interpreters and Compilers
3. Editors
4. DBMS
5. CAD/CAM applications
6. Word Processors etc…

C Tokens:-

The basic and the smallest units of a C program are called C tokens. There are six types of tokens in C. They are:

1. Keywords
2. Identifiers
3. Constants
4. Strings
5. Operators
6. Special Symbols

Keywords and Identifiers:-

Every word in a C program is either a keyword or identifier. All keywords are basically the sequences of characters that have one or more fixed meanings and these meanings in any circumstances can't be changed. All C keywords must be written in lower letters.

Keywords supported by ANSI C are:

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

Identifiers are the names given to the program elements such as variables, arrays and functions. Basically, identifiers are the sequences of alphabets and digits. The rules that govern the formation of identifier names are:

1. The first character must be an alphabet or an underscore.
2. All succeeding characters must be either letters or digits
3. Uppercase and lowercase identifiers are different
4. No special character or punctuation symbols are allowed except the underscore.
5. No two successive underscores are allowed.
6. Keywords should not be used as identifiers.

Basic Structure of a C program:-

Different programming languages have their own format of coding. The basic components of a C program are:

a. main()
b. A pair of curly braces { }
c. Declarations and statements
d. User defined functions

The complete structure of a C program:

```
preprocessor statements
global declarations
main()
{
        declarations;
        statements;
}
User defined functions
```

Preprocessor statements:-

These statements begin with # symbol and also called as preprocessor directives. These statements direct the C preprocessor to include header files and also symbolic constants into a C program.

Examples:-

#include<stdio.h>	-→	For the standard input / output functions
#include "Test.h"	-→	for the inclusion of header file Test.
#define NULL 0	-→	for defining symbolic constant, NULL=0

Global declarations:-

Variables or functions whose existence is known in the main() function and other user defined functions, are called the global variables or functions and their declarations are called global declarations.

This declarations should be made before main().

The main() function:-

As the name itself indicates, this is the main function of every C program. Execution of C program starts with main(). No C program is executed without the main() function. The function main() should be written in lowercase letters and should not be terminated by a semicolon. There must be one and only one main() function in every C program.

Braces:-

Every C program uses a pair of curly braces ({, }). The left brace indicates the beginning of the main() function. The right brace indicates the end of the main() function. The braces can also be used to indicate the beginning and end of use-defined functions.

Declarations:-

The declaration is a part of the C program where all the variables, arrays, functions etc.. used in the C program are declared and may be initialized with their basic data types.

Statements:-

These are instructions to the computer to perform some specific operations. They may be I/O statements, arithmetic statements, control statements and other statements. They also include comments. The comments are explanatory notes on some instructions. The comment statements are not compiled and executed.

User defined functions:-

These are subprograms. A subprogram is a function. The user-defined functions contain a set of statements to perform a specific task. These are written by the user. They may be written before or after the main() function.

Example:

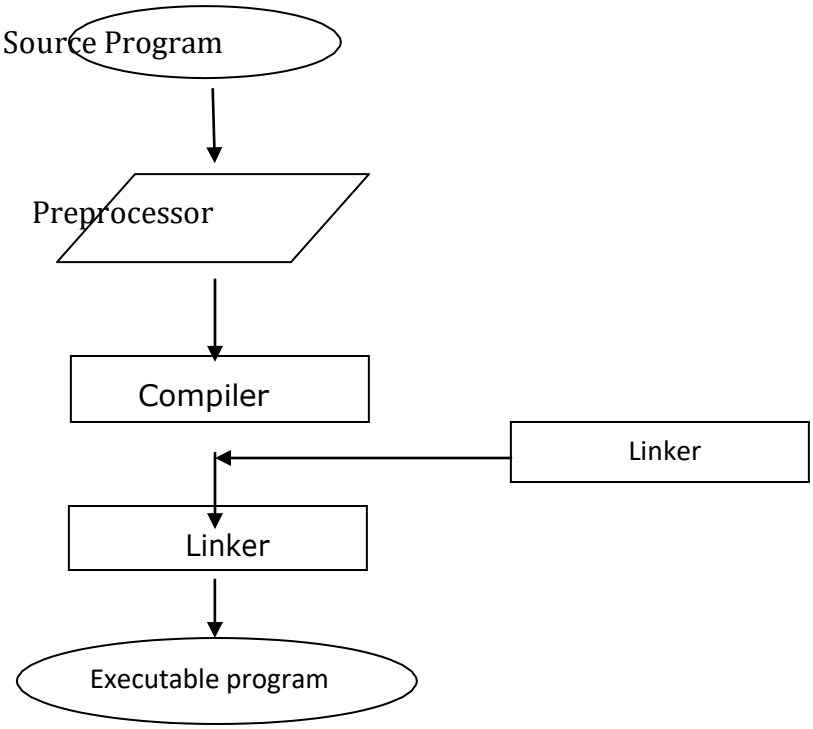
```
#include <stdio.h>
void main()
{
        printf("Welcome to C \n");
}
```

Compiling and Executing a C program:-

Compiling a C program means translating it into machine language. C compilers are used for this purpose. The C program to be compiled must be typed in using an editor. An editor is a program which allows the programmer to write the program and edit it. C compilers are available with and without editors.

The environment where you find the compiler, editor, debugging tools, linking facilities and testing tools is called the Integrated Development Environment (IDE).

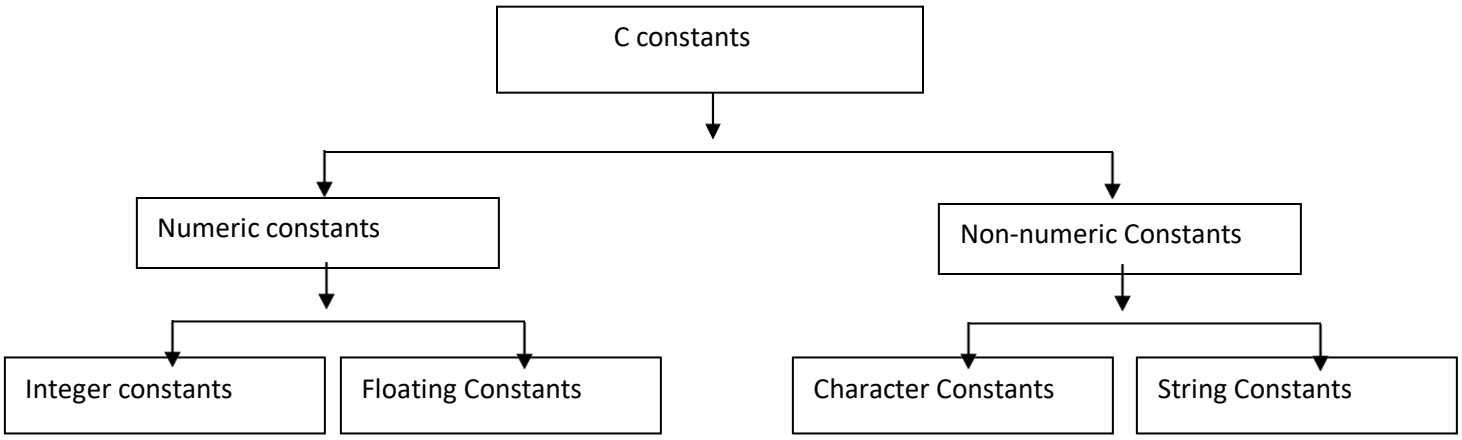
Examples:- Turbo C, Borland C, Microsoft and ANSI C



Constants and Variables:-

The data processing requires a set of data to be entered or displayed. This data may be a number or string of chars. The programmer will decide which type data is required to solve the given problem.

The quantity which does not change during the execution of a program is known as constant. There are two types of constants:



1. An integer constant is a whole number. It is a sequence of digits without a decimal point. It is prefixed with plus or minus sign.
2. A floating constant is a number with decimal point. It is defined as a sequence of digits preceded and followed by a decimal point. It is prefixed with plus or minus sign.
3. A character constant is a single character enclosed within a pair of single quote.
4. A string constant is a sequence of characters enclosed within a pair of double quotes.

Variables:-

Variables are nothing but reserved memory locations to store values. This means when you create a variable, you reserve some space in memory. They are used to denote constants, functions, arrays, fields of structures, name of files etc..

Rules for forming variable names:-

1. The first character must be an alphabet or an underscore.
2. All succeeding characters must be either letters or digits
3. Uppercase and lowercase variables are different
4. No special character or punctuation symbols are allowed except the underscore.
5. No two successive underscores are allowed.
6. Keywords should not be used as variables.

Declaration of Variables:-

All variables must be declared before they are used in C program. The purpose of declaring variables is to reserve the amount of memory required for these variables.

Syntax for declaration:-

Data_type variable_name semicolon

Data_type:- is a basic data type such as int, float, char or double.

Example: **int length;**

Assigning values to variables:

When the data is stored that the variables represent some memory location. Each variable is associated with one or more values. The process of giving values to variables is called the assignment of the values. The assignment operator '=' is used to assign a value to a variable.

Syntax:

Variable_name = value;

Example: int x = 1;

Data types:-

Data types indicate the type of data that a variable can hold. The data may be numeric or non-numeric. Those are categorized into:

1. Built in data types
2. Derived data types
3. User defined data types

1.Built in Data types:- They are basic or primitive data types and designate a single value. There are four fundamental built in data types in C language. They are:

1. Integer
2. Floating point number
3. Double real number
4. Characters

In C, there is one and only one non-specific data type called void. It does not specify anything.

Types	**Keyword**	**Size (in bytes)**
Integer	Int	2
Floating point	Float	4
Double precision	Double	8
Character	Char	1
Non-specific	Void	__

Int:-

Int is a keyword used to indicated integer number. Any integer number is a sequence of digits without decimal point.

Float:-

Float is a keyword used to indicate a floating point number. The floating point numbers are the same as real numbers. They are called floating point numbers because the decimal point is shifted either to left or to the right of some digits during manipulation.

Char:-

Char is a keyword used to indicate the character type data. The data may be a character constant or a string constant. A character constant can be defined as any single character enclosed in a pair of single quotes.

Double:-

Double is a keyword used to indicate a double precision floating point number. The precision associated with the accuracy of the data. It is used to whenever more accuracy is required in representing the floating point numbers. A double is normal float value, but the number of significant digits that are stored after the decimal point is double that of the float.

The float usually stores a maximum of 6 digits after the decimal point but double stores 16 significant digits after the decimal point.

Input and Output Functions:-

Every computer program takes some data as input and prints the processed data. The data can be provided in two ways.

The first one is known as non-interactive method in which we assign values to the variables.

The second method known as an interactive input method where the data is supplied by the user through the standard input device (keyboard).

C is a functional programming language. So, it has no exclusive built-in functions to perform the basic input-output operations. To perform these input-output operations C provides a library functions. This library is called a standard input-output library. It is denoted by stdio. The header file containing such library functions called stdio.h.

Some of the standard input-output functions in C:

1. scanf()
2. printf()
3. getchar()
4. putchar()

5. gets()
6. puts()
7. getch()
8. getche()

There are two types of input-output (I/O) functions. They are:

1. formatted I/O functions
2. Unformatted I/O functions

Operators

C programming has various operators to perform tasks including arithmetic, conditional and bitwise operations.

An operator is a symbol which operates on a value or a variable. For example:+ is an operator to perform addition on operands.

C programming has wide range of operators to perform various operations.

Operators in C programming:-

1. Arithmetic Operators
2. Increment and Decrement Operators
3. Assignment Operators
4. Relational Operators
5. Logical Operators
6. Conditional Operators
7. Bitwise Operators
8. Special Operators
    a. Shift Operators
    b. Comma Operator
    c. Sizeof Operator
    d. Ternary Operator

1. Arithmetic Operators:-

An arithmetic operator performs mathematical operations such as addition, subtraction and multiplication on numerical values (constants and variables).

Operator	Meaning of Operator
+	addition or unary plus
-	subtraction or unary minus
*	multiplication
/	division

% remainder after division(modulo division)

Example #1: Arithmetic Operators

```
// C Program to demonstrate the working of arithmetic operators
#include <stdio.h>
int main()
{
   int a = 9,b = 4, c;
   c = a+b;
   printf("a+b = %d \n", c);
   c = a-b;
   printf("a-b = %d \n", c);
   c = a*b;
   printf("a*b = %d \n", c);
   c=a/b;
   printf("a/b = %d \n", c);
   c=a%b;
   printf("Remainder when a divided by b = %d \n", c);
   return 0;
}
```

Output:-

```
a+b = 13
a-b = 5
a*b = 36
a/b = 2
Remainder when a divided by b=1
```

The operators +, - and * computes addition, subtraction and multiplication respectively as you might have expected.

In normal calculation, 9/4 = 2.25. However, the output is 2 in the program. It is because both variables a and b are integers. Hence, the output is also an integer. The compiler neglects the term after decimal point and shows answer 2 instead of 2.25.

The modulo operator % computes the remainder. When a = 9 is divided by b = 4, the remainder is 1. The % operator can only be used with integers.

2. **<u>Increment and decrement operators:-</u>**

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 #2: Increment and Decrement Operators

// C Program to demonstrate the working of increment and decrement operators

```
#include <stdio.h>
int 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);
   return 0;
}
```

Output

```
++a = 11
--b = 99
++c = 11.500000
++d = 99.500000
```

Here, the operators ++ and -- are used as prefix. These two operators can also be used as postfix like a++ and a--.

++ and -- operator as prefix and postfix:-

Suppose you use ++ operator as prefix like: **++var**. The value of ***var*** is incremented by 1 then, it returns the value.

Suppose you use ++ operator as postfix like: **var++**. The original value of ***var*** is returned first then, ***var*** is incremented by 1.

Example:- C program to demonstrate post increment and pre increment.

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

```
 int var=5;
 // 5 is displayed then, var is increased to 6.
 printf("%d\n",var++);
 // Initially, var = 6. It is increased to 7 then, it is displayed.
 printf("%d",++var);
 return 0;
}
```

Output

5

7

3. C Assignment Operators:-

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

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 #3: Assignment Operators

```
// C Program to demonstrate the working of assignment operators
#include <stdio.h>
int main()
{
   int a = 5, c;
   c = a;
   printf("c = %d \n", c);
   c += a; // c = c+a
   printf("c = %d \n", c);
   c -= a; // c = c-a
   printf("c = %d \n", c);
```

```
    c *= a; // c = c*a
    printf("c = %d \n", c);
    c /= a; // c = c/a
    printf("c = %d \n", c);
    c %= a; // c = c%a
    printf("c = %d \n", c);
    return 0;
}
```

Output

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

4. **C 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.

Operator	Meaning of Operator	Example
==	Equal to	5 == 3 returns 0
>	Greater than	5 > 3 returns 1
<	Less than	5 < 3 returns 0
!=	Not equal to	5 != 3 returns 1
>=	Greater than or equal to	5 >= 3 returns 1
<=	Less than or equal to	5 <= 3 return 0

Example #4: Relational Operators

```
// C Program to demonstrate the working of arithmetic operators
#include <stdio.h>
int main()
{
    int a = 5, b = 5, c = 10;
    printf("%d == %d = %d \n", a, b, a == b); // true
    printf("%d == %d = %d \n", a, c, a == c); // false
```

```
    printf("%d > %d = %d \n", a, b, a > b); //false
    printf("%d > %d = %d \n", a, c, a > c); //false
    printf("%d < %d = %d \n", a, b, a < b); //false
    printf("%d < %d = %d \n", a, c, a < c); //true
    printf("%d != %d = %d \n", a, b, a != b); //false
    printf("%d != %d = %d \n", a, c, a != c); //true
    printf("%d >= %d = %d \n", a, b, a >= b); //true
    printf("%d >= %d = %d \n", a, c, a >= c); //false
    printf("%d <= %d = %d \n", a, b, a <= b); //true
    printf("%d <= %d = %d \n", a, c, a <= c); //true
    return 0;
}
```

Output

```
5 == 5 = 1
5 == 10 = 0
5 > 5 = 0
5 > 10 = 0
5 < 5 = 0
5 < 10 = 1
5 != 5 = 0
5 != 10 = 1
5 >= 5 = 1
5 >= 10 = 0
5 <= 5 = 1
5 <= 10 = 1
```

5. C 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 of Operator	Example
&&	Logical AND True only if all operands are true	If c=5 and d=2 then, (c==5) && (d>5) equals to 0.

||	Logical OR True only if either one operand is true	If c=5 and d=2 then, (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.

Example #5: Logical Operators

```
// C Program to demonstrate the working of logical operators
#include <stdio.h>
int main()
{
   int a = 5, b = 5, c = 10, result;
   result = (a = b) && (c > b);
   printf("(a = b) && (c > b) equals to %d \n", result);
   result = (a = b) && (c < b);
   printf("(a = b) && (c < b) equals to %d \n", result);
   result = (a = b) || (c < b);
   printf("(a = b) || (c < b) equals to %d \n", result);
   result = (a != b) || (c < b);
   printf("(a != b) || (c < b) equals to %d \n", result);
   result = !(a != b);
   printf("!(a == b) equals to %d \n", result);
   result = !(a == b);
   printf("!(a == b) equals to %d \n", result);
   return 0;
}
```

Output

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

Explanation of logical operator program

1. (a = b) && (c > 5) evaluates to 1 because both operands (a = b) and (c > b) is 1 (true).
2. (a = b) && (c < b) evaluates to 0 because operand (c < b) is 0 (false).
3. (a = b) || (c < b) evaluates to 1 because (a = b) is 1 (true).
4. (a != b) || (c < b) evaluates to 0 because both operand (a != b) and (c < b) are 0 (false).
5. !(a != b) evaluates to 1 because operand (a != b) is 0 (false). Hence, !(a != b) is 1 (true).
6. !(a == b) evaluates to 0 because (a == b) is 1 (true). Hence, !(a == b)is 0 (false).

6. Bitwise Operators

In processor, mathematical operations like: addition, subtraction, addition and division are done in bit-level which makes processing faster and saves power.

Bitwise operators are used in C programming to perform bit-level operations.

Operators	Meaning of operators
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise complement
<<	Shift left
>>	Shift right

Bitwise AND operator &

The output of bitwise AND is 1 if the corresponding bits of all operands is 1. If either bit of an operand is 0, the result of corresponding bit is evaluated to 0.

Let us suppose the bitwise AND operation of two integers 12 and 25.

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bit Operation of 12 and 25

```
 00001100
& 00011001
_________
 00001000 = 8 (In decimal)
```

Example #1: Bitwise AND

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

```
    int a = 12, b = 25;
    printf("Output = %d", a&b);
    return 0;
}
```

Output

```
Output = 8
```

Bitwise OR operator |

The output of bitwise OR is 1 if at least one corresponding bit of two operands is 1. In C Programming, bitwise OR operator is denoted by |.

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bitwise OR Operation of 12 and 25

```
 00001100
| 00011001
 ________
 00011101 = 29 (In decimal)
```

Example #2: Bitwise OR

```
#include <stdio.h>
int main()
{
    int a = 12, b = 25;
    printf("Output = %d", a|b);
    return 0;
}
```

Output

```
Output = 29
```

Bitwise XOR (exclusive OR) operator ^

The result of bitwise XOR operator is 1 if the corresponding bits of two operands are opposite. It is denoted by ^.

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bitwise XOR Operation of 12 and 25

```
 00001100
| 00011001
```

00010101 = 21 (In decimal)

Example #3: Bitwise XOR

```
#include <stdio.h>
int main()
{
   int a = 12, b = 25;
   printf("Output = %d", a^b);
   return 0;
}
```

Output

```
Output = 21
```

Bitwise complement operator ~

Bitwise compliment operator is an unary operator (works on one operand only). It changes the 1 to 0 and 0 to 1. It is denoted by ~.

35=00100011 (In Binary)

Bitwise complement Operation of 35

~ 00100011

11011100 = 220 (In decimal)

For any integer n, bitwise complement of n will be -(n+1). To understand this, you should have the knowledge of 2's complement.

2's Complement

Two's complement is an operation on binary numbers. The 2's complement of a number is equal to the complement of that number plus 1.

For example:

Decimal	Binary	2's complement
0	00000000	-(11111111+1) = -00000000 = -0(decimal)
1	00000001	-(11111110+1) = -11111111 = -256(decimal)
12	00001100	-(11110011+1) = -11110100 = -244(decimal)
220	11011100	-(00100011+1) = -00100100 = -36(decimal)

Bitwise complement of any number N is -(N+1).

Example #4: Bitwise complement

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

```
int main()
{
   printf("complement=%d\n",~35);
   printf("complement=%d\n",~-12);
   return 0;
}
```

Output

```
complement=-36
complement=11
```

Shift Operators in C programming

There are two shift operators in C programming:

1. Right shift operator
2. Left shift operator.

Right Shift Operator

Right shift operator shifts all bits towards right by certain number of specified bits. It is denoted by >>.

212 = 11010100 (In binary)

212>>2 = 00110101 (In binary) [Right shift by two bits]

212>>7 = 00000001 (In binary)

212>>8 = 00000000

212>>0 = 11010100 (No Shift)

Left Shift Operator

Left shift operator shifts all bits towards left by certain number of specified bits. It is denoted by <<.

212 = 11010100 (In binary)

212<<1 = 110101000 (In binary) [Left shift by one bit]

212<<0 =11010100 (Shift by 0)

212<<4 = 110101000000 (In binary) =3392(In decimal)

Example #5: Shift Operators

```
#include <stdio.h>
int main()
{
   int num=212, i;
   for (i=0; i<=2; ++i){
```

```
    printf("Right shift by %d: %d\n", i, num>>i);
        }
    printf("\n");

    for (i=0; i<=2; ++i) {
      printf("Left shift by %d: %d\n", i, num<<i);
        }
    return 0;
}
```

Output:

```
Right Shift by 0: 212
Right Shift by 1: 106
Right Shift by 2: 53

Left Shift by 0: 212
Left Shift by 1: 424
Left Shift by 2: 848
```

Comma Operator

Comma operators are used to link related expressions together. For example:

```
int a, c = 5, d;
```

The sizeof operator

The sizeof is an unary operator which returns the size of data (constant, variables, array, structure etc).

Example #6: sizeof Operator implementation in C

```
#include <stdio.h>
int main()
{
   int a, e[10];
   float b;
   double c;
   char d;
   printf("Size of int=%lu bytes\n",sizeof(a));
   printf("Size of float=%lu bytes\n",sizeof(b));
   printf("Size of double=%lu bytes\n",sizeof(c));
```

```
    printf("Size of char=%lu byte\n",sizeof(d));
    printf("Size of integer type array having 10 elements = %lu bytes\n", sizeof(e));
    return 0;
}
```

Output

```
Size of int = 4 bytes
Size of float = 4 bytes
Size of double = 8 bytes
Size of char = 1 byte
Size of integer type array having 10 elements = 40 bytes
```

C Ternary Operator (?:)

A conditional operator is a ternary operator, that is, it works on 3 operands.

Conditional Operator Syntax

conditionalExpression ? expression1 : expression2

The conditional operator works as follows:

1. The first expression conditionalExpression is evaluated at first. This expression evaluates to 1 if it's and evaluates to 0 if it's false.
2. If conditionalExpression is true, expression1 is evaluated.
3. If conditionalExpression is false, expression2 is evaluated.

Example #6: C conditional Operator

```
#include <stdio.h>
int main(){
  char February;
  int days;
  printf("If this year is leap year, enter 1. If not enter any integer: ");
  scanf("%c",&February);
  // If test condition (February == 'l') is true, days equal to 29.
  // If test condition (February =='l') is false, days equal to 28.
  days = (February == '1') ? 29 : 28;
  printf("Number of days in February = %d",days);
  return 0;
}
```

Output

```
If this year is leap year, enter 1. If not enter any integer: 1
```

Number of days in February = 29

if else:

The if statement in C language is used to perform operation on the basis of condition. By using if-else statement, you can perform operation either condition is true or false.

There are many ways to use if statement in C language:

- If statement
- If-else statement
- If else-if ladder

If Statement

The single if statement in C language is used to execute the code if condition is true. The syntax of if statement is given below:

```
if(expression){
        //code to be executed
}
```

Flowchart of if statement in C

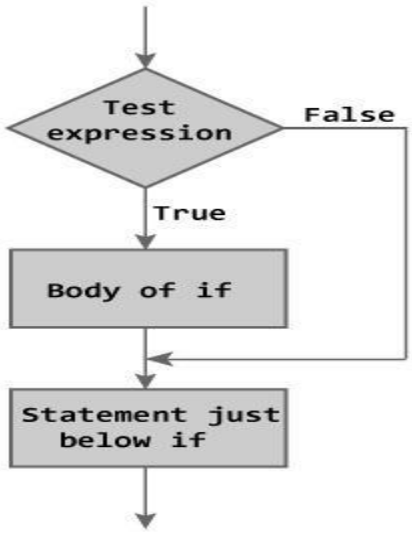


Figure: Flowchart of if Statement

Example:-

```
#include<stdio.h>
#include<conio.h>
void main()
{
int number=0;
```

```
clrscr();
printf("enter a number:");
scanf("%d",&number);
if(number%2==0){
        printf("%d is even number",number);
    }
    getch();
    }
```

Output

```
enter a number:4
4 is even number
```

If-else Statement

The if-else statement in C language is used to execute the code if condition is true or false. The syntax of if-else statement is given below:

```
if(expression){
    //code to be executed if condition is true
}
else{
    //code to be executed if condition is false
}
```

Flowchart of if-else statement in C

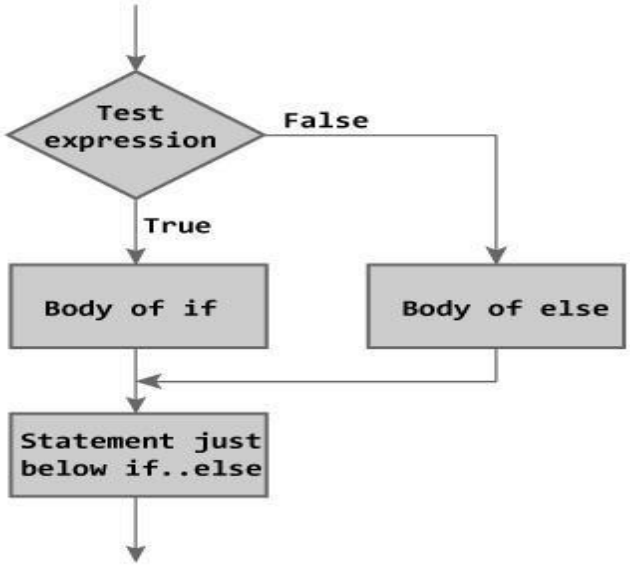


Figure: Flowchart of if...else Statement

Let's see the simple example of even and odd number using if-else statement in C language.

Example:-

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

Output

```
Enter a number: 4
4 is even number
Enter a number: 5
5 is odd number
```

If else-if ladder Statement

The if else-if statement is used to execute one code from multiple conditions. The syntax of if else-if statement is given below:

```
if(condition1){
        //code to be executed if condition1 is true
}
else if(condition2){
        //code to be executed if condition2 is true
}
else if(condition3){
        //code to be executed if condition3 is true
}
...
```

```
Else{
        //code to be executed if all the conditions are false
}
```

Flowchart of else-if ladder statement in C

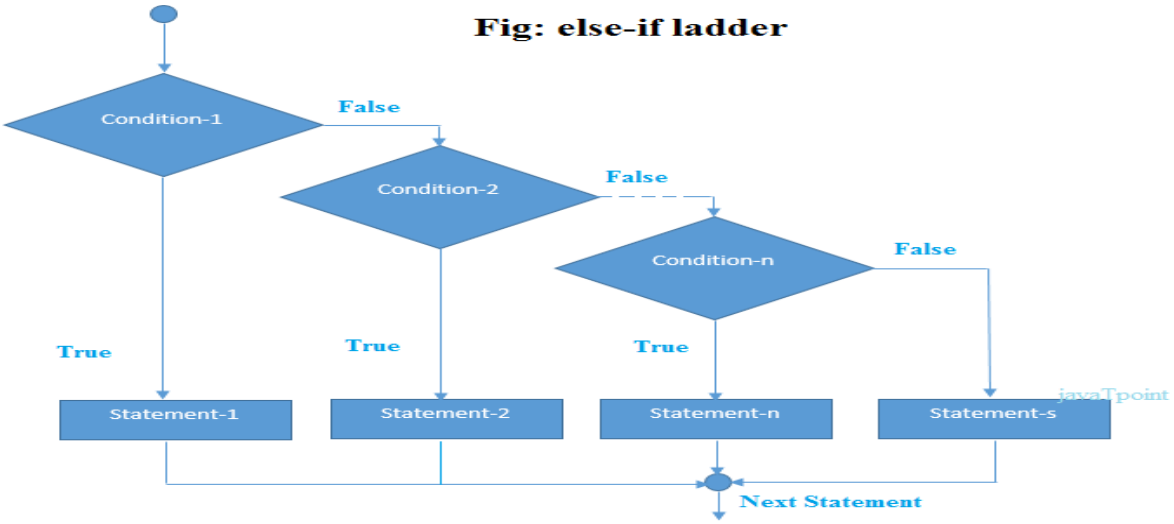


Fig: else-if ladder

The example of if-else-if statement in C language is given below.

Example:

```
#include<stdio.h>
#include<conio.h>
void main(){
int number=0;
clrscr();
printf("enter a number:");
scanf("%d",&number);
if(number==10){
printf("number is equals to 10");
}
else if(number==50){
printf("number is equal to 50");
}
else if(number==100){
printf("number is equal to 100");
}
else{
printf("number is not equal to 10, 50 or 100");
```

```
}
getch();
}
```

Output

```
enter a number:4
number is not equal to 10, 50 or 100
enter a number:50
number is equal to 50
```

C Switch Statement

The switch statement in C language is used to execute the code from multiple conditions. It is like if else-if ladder statement.

The syntax of switch statement in c language is given below:

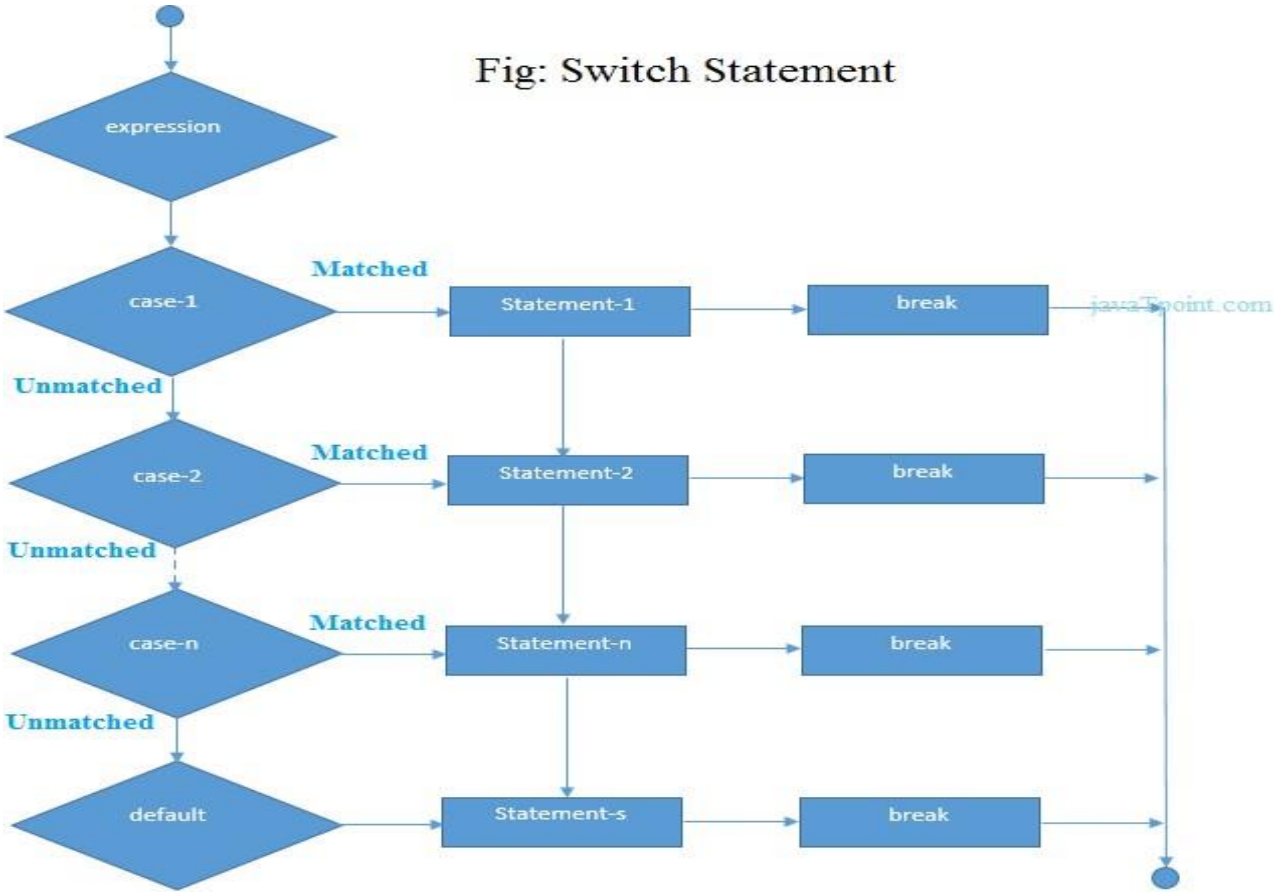
```
switch(expression){
case value1:
    //code to be executed;
    break; //optional
case value2:
    //code to be executed;
    break; //optional
.....

default:
    code to be executed if all cases are not matched;
}
```

Rules for switch statement in C language

- The switch expression must be of integer or character type.
- The case value must be integer or character constant.
- The case value can be used only inside the switch statement.
- The break statement in switch case is not must. It is optional. If there is no break statement found in switch case, all the cases will be executed after matching the case value. It is known as fall through state of C switch statement.

Flowchart of switch statement in C



Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int number=0;
clrscr();
printf("enter a number:");
scanf("%d",&number);
switch(number)
{
case 10:
        printf("number is equals to 10");
        break;
case 50:
        printf("number is equal to 50");
```

```
        break;
    case 100:
        printf("number is equal to 100");
        break;
    default:
        printf("number is not equal to 10, 50 or 100");
    }
    getch();
    }
```

Output

```
enter a number:4
number is not equal to 10, 50 or 100
enter a number:50
number is equal to 50
```

C Switch statement is fall-through

In C language, switch statement is fall through, it means if you don't use break statement in switch case, all the case after matching case will be executed.

Let's try to understand the fall through state of switch statement by the example given below.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int number=0;
clrscr();
printf("enter a number:");
scanf("%d",&number);
switch(number)
{
case 10:
    printf("number is equals to 10\n");
case 50:
    printf("number is equal to 50\n");
case 100:
```

```
            printf("number is equal to 100\n");
        default:
            printf("number is not equal to 10, 50 or 100");
        }
        getch();
        }
```

Output

```
enter a number:10
number is equals to 10
number is equals to 50
number is equals to 100
number is not equal to 10, 50 or 100

enter a number:50
number is equal to 50
number is equals to 100
number is not equal to 10, 50 or 100
```

Goto statement:

The goto statement is used to alter the normal sequence of a C program.

Syntax:

```
Goto label;
…. ….. ……..
……..……
Label;
Statement;
```

The label is an identifier. When goto statement is encountered, control of the program jumps to label: and starts executing the code.

```
Goto label;
…. ….. ……..
……..……
Label;
```

→ …. ….. ……..

……..

Example:

// Program to calculate the sum and average of maximum of 5 numbers

// If user enters negative number, the sum and average of previously entered // positive number is displayed

```
# include <stdio.h>

int main()
{
   const int maxInput = 5;
   int i;
   double number, average, sum=0.0;
   for(i=1; i<=maxInput; ++i)
   {
      printf("%d. Enter a number: ", i);
      scanf("%lf",&number);

// If user enters negative number, flow of program moves to label jump
      if(number < 0.0)
         goto jump;

      sum += number; // sum = sum+number;
   }
   jump:
   average=sum/(i-1);
   printf("Sum = %.2f\n", sum);
   printf("Average = %.2f", average);
   return 0;
}
```

Output

1. Enter a number: 3
2. Enter a number: 4.3

3. Enter a number: 9.3

4. Enter a number: -2.9

Sum = 16.60

Looping controls Structures:

Loops are used in programming to repeat a specific block until some end condition is met. There are three loops in C programming:

1. for loop
2. while loop
3. do...while loop

for Loop

The "for loop" loops from one number to another number and increases by a specified value each time.

Syntax:

```
for (initializationStatement; testExpression; updateStatement)
{
    // codes
}
```

- The initialization statement is executed only once.
- Then, the test expression is evaluated. If the test expression is false (0), for loop is terminated. But if the test expression is true (nonzero), codes inside the body of **for** loop is executed and the update expression is updated. This process repeats until the test expression is false.
- The **for** loop is commonly used when the number of iterations is known.

for loop Flowchart

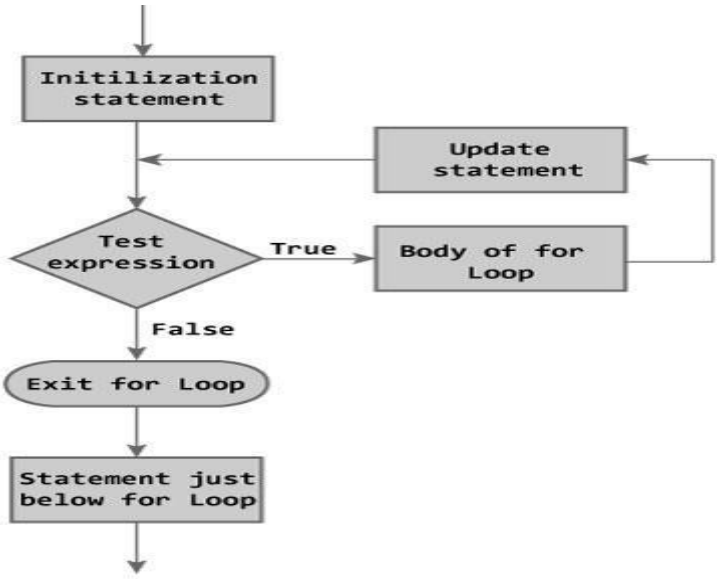


Figure: Flowchart of for Loop

Example: for loop

Write a Program to calculate the sum of first n natural numbers and Positive integers 1,2,3...n are known as natural numbers?

```
#include <stdio.h>
void main()
{
   int n, count, sum = 0;
   printf("Enter a positive integer: ");
   scanf("%d", &n);
   for(count = 1; count <= n; ++count) // for loop terminates when n is less than count
   {
      sum += count;
   }
   printf("Sum = %d", sum);
}
```

Output

```
Enter a positive integer: 10
Sum = 55
```

Explanation:

The value entered by the user is stored in variable **n**. Suppose the user entered 10.

The **count** is initialized to 1 and the test expression is evaluated. Since, the test expression **count <= n** (1 less than or equal to 10) is true, the body of for loop is executed and the value of **sum** will be equal to 1.

Then, the update statement **++count** is executed and count will be equal to 2. Again, the test expression is evaluated. The test expression is evaluated to true and the body of for loop is executed and the **sum** will be equal to 3. And, this process goes on.

Eventually, the count is increased to 11. When the count is 11, the test expression is evaluated to 0 (false) and the loop terminates.

While loop:-

The while loop can be used if you don't know how many times a loop must run.

Syntax:

```
while (testExpression)
{
  //codes
}
```

- The while loop evaluates the test expression. If the test expression is true (nonzero), codes inside the body of while loop is evaluated. Then, again the test expression is evaluated. The process goes on until the test expression is false.
- When the test expression is false, the while loop is terminated.

Flowchart of while loop

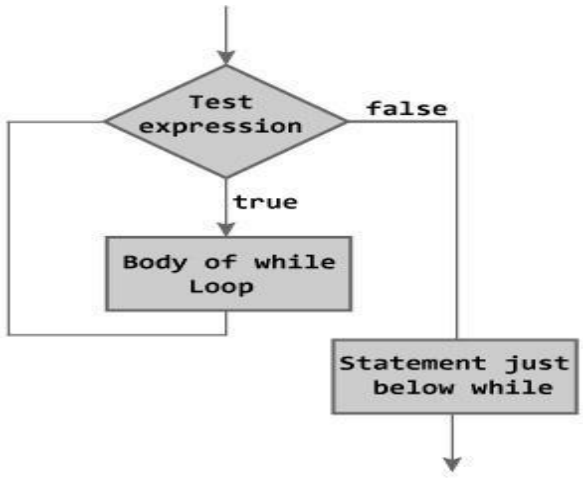


Figure: Flowchart of while Loop

Example #1: while loop

Write a program to find factorial of a number ?

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

```
void main()
{
   int number;
   long long factorial;
   printf("Enter an integer: ");
   scanf("%d",&number);
   factorial = 1;
   // loop terminates when number is less than or equal to 0
   while (number > 0)
   {
      factorial *= number; // factorial = factorial*number;
      --number;
   }
   printf("Factorial= %lld", factorial);
}
```

Output

```
Enter an integer: 5
Factorial = 120
```

do...while loop

The **do..while** loop is similar to the **while** loop with one important difference. The body of **do...while** loop is executed once, before checking the test expression. Hence, the **do...while loop** is executed at least once.

do...while loop Syntax

```
do
{
  // codes
}
while (testExpression);
```

- The code block (loop body) inside the braces is executed once. Then, the test expression is evaluated. If the test expression is true, the loop body is executed again. This process goes on until the test expression is evaluated to 0 (false).
- When the test expression is false (nonzero), the **do...while** loop is terminated.

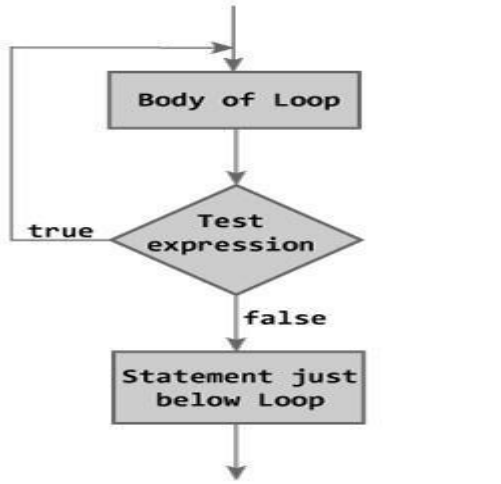


Figure: Flowchart of do...while Loop

Example #2: do...while loop

```
// Program to add numbers until user enters zero
        #include <stdio.h>
        Void main()
        {
           double number, sum = 0;
           // loop body is executed at least once
           do
           {
              printf("Enter a number: ");
              scanf("%lf", &number);
              sum += number;
           }
           while(number != 0.0);
           printf("Sum = %.2lf",sum);
        }
```

Output

```
Enter a number: 1.5
Enter a number: 2.4
Enter a number: -3.4
Enter a number: 4.2
Enter a number: 0
Sum = 4.70
```

Exercises:-

1. Write a program to print the first 5 numbers with their squares and cubes using for loop?
2. Write a program to print five entered numbers in Ascending Order using for loop?
3. Write a program to calculate factorial of a given number using while loop?
4. Write a program to compute the factorial of a given number using do-while loop?
5. Write a program to print the ASCII equivalent of each character of an input string using while?

Break and continue:

It is sometimes desirable to skip some statements inside the loop or terminate the loop immediately without checking the test expression. In such cases, **break** and **continue** statements are used.

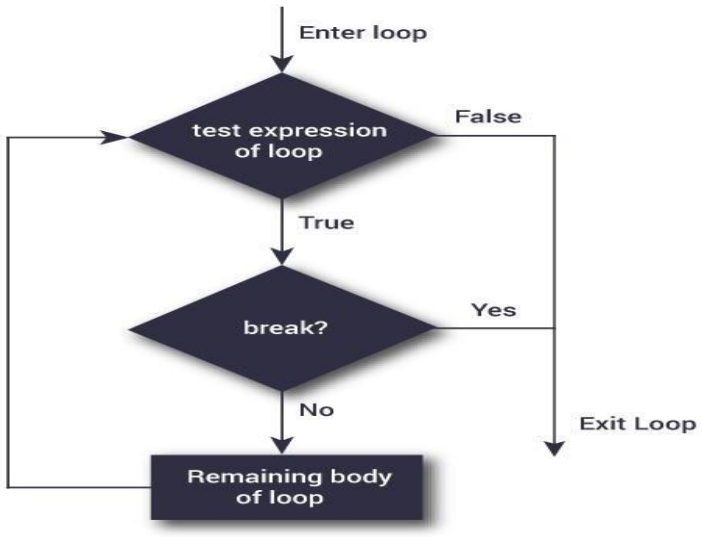
break Statement

The break statement terminates the loop immediately when it is encountered. The break statement is used with decision making statement such as if...else.

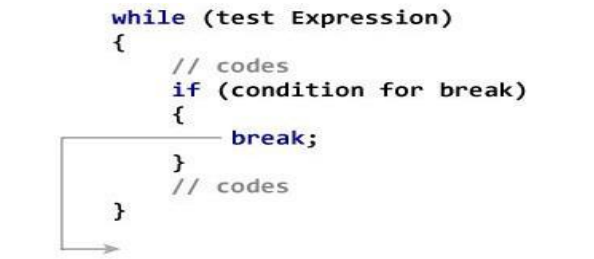
Syntax of break statement

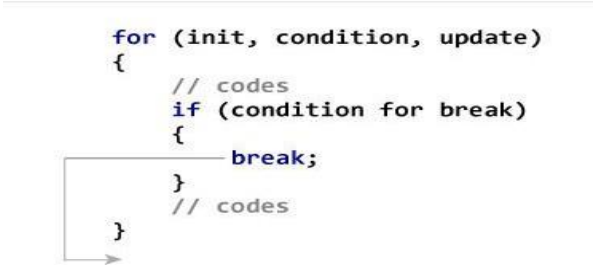
```
break;
```

Flowchart of break statement



Working of Break statement:





Example #1: break statement

// Program to calculate the sum of maximum of 10 numbers

```
# include <stdio.h>
int main()
{
   int i;
   double number, sum = 0.0;

   for(i=1; i <= 10; ++i)
   {
      printf("Enter a n%d: ",i);
      scanf("%lf",&number);

      if(number < 0.0) // If user enters negative number, loop is terminated
      {
         break;
      }
      sum += number;
   }
   printf("Sum = %.2f",sum);
```

```
}
```

Output

```
Enter a n1: 2.4
Enter a n2: 4.5
Enter a n3: 3.4
Enter a n4: -3
Sum = 10.30
```

Explanation: This program calculates the sum of maximum of 10 numbers. It's because, when the user enters negative number, the break statement is executed and loop is terminated.

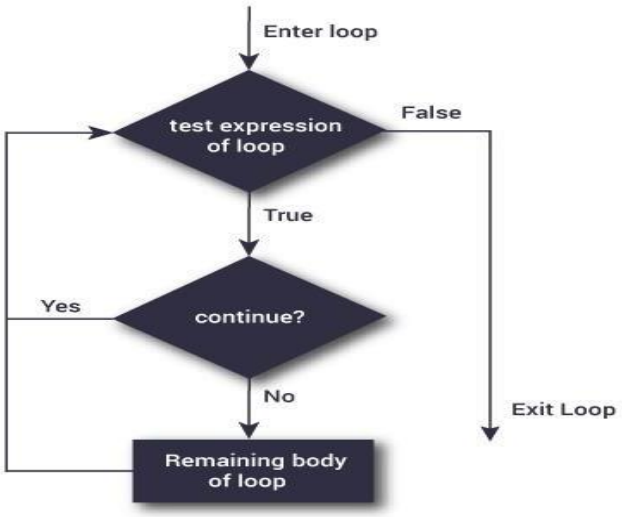
Continue Statement

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

Syntax of continue Statement

```
continue;
```

Flowchart of continue Statement



Continue statement working:

```
while (test Expression)
{
      // codes
      if (condition for continue)
      {
            continue;
      }
      // codes
}
```

```
for (init, condition, update)
{
      // codes
      if (condition for continue)
      {
            continue;
      }
      // codes
}
```

Example #2: continue statement

// Program to calculate sum of maximum of 10 numbers and Negative numbers are skipped from calculation

```
# include <stdio.h>
void main()
{
   int i;
   double number, sum = 0.0;
```

```
    for(i=1; i <= 10; ++i)
    {
        printf("Enter a n%d: ",i);
        scanf("%lf",&number);

        // If user enters negative number, loop is terminated
        if(number < 0.0)
        {
            continue;
        }
        sum += number;
    }
    printf("Sum = %.2lf",sum);
}
```

Output

```
Enter a n1: 1.1
Enter a n2: 2.2
Enter a n3: 5.5
Enter a n4: 4.4
Enter a n5: -3.4
Enter a n6: -45.5
Enter a n7: 34.5
Enter a n8: -4.2
Enter a n9: -1000
Enter a n10: 12
Sum = 59.70
```

In the program, when the user enters positive number, the sum is calculated using **sum += number;** statement.

When the user enters negative number, the **continue** statement is executed and skips the negative number from calculation.

ARRAYS

Array Definition

An array can be defined as ordered list of same type of data elements. These elements may be data type int, float, char or double. All these elements are stored in consecutive memory location.

An array is described by a single name or identifier. And each element in an array is referenced by subscript (**an index**) enclosed in a pair of square brackets. This subscript indicates the position of an individual data item in an array. The index must be positive integer.

Characteristic features of an array

1. Array is a data structure storing a group of elements, all of which are of the same data type.
2. All the elements of an array share the same name, and they are distinguished from one another with the help of an index.
3. Random access to every element using a numeric index of an array.

Classification of array

Array are classified into two types:

1. One dimensional arrays
2. Multi dimensional array

Declaration of an array

An array must be declared before it appears in a c program. This is done using an appropriate data type. At the same time the size of an array must be specified .This allow the compiler to decide on how much memory has to be reserved for the given array.

Syntax of an array

```
data_type  arrayname[size];
```

data type : any basic data type or a user defined data type

array name: is the name of an array

size: number of elements of type data_type. And the size must be an integer constant specified within a pair of square brackets.

1. One dimensional array:

This is a linear list of a fixed number of data items of the same type. All these data items are accessed using the same name and a single subscript. It is similar to row or a column matrix. It is also called a single dimensional array or a one subscripted variable.

Example

```
int list[10];
char name[20];
```

```
float xyz[5];
double p[100];
```

Rules for subscripts

1. Each subscript must be an unsigned positive integer constant or an expression
2. Subscript of subscript is not allowed
3. The maximum subscript appearing in a program for a subscripted variable should not exceed the declared one.
4. The subscript values ranges from 0 to one less than maximum size. If the size of an array is 10, the first subscript is 0, the second subscript is 1 and so on the last subscript is 9.

Total size of array:-

The total amount of memory that can be allocated to a one-dimensional array is computed as, size*[sizeof(data_type)].

Syntax for calculating total size of array:

```
Total_size=size*[sizeof (data_type)];
```

size is a number of elements of a one dimensional array

sizeof() is an unary operator to find the size in bytes

data_type basic data type or a user-defined data type

Example:-

```
#include <stdio.h>
void main()
{
   int number[10];
   printf("The total size of array is : %d\n",sizeof(number));
}
```

The total size of a one-dimensional array of 10 integers is 40 bytes. Because the size of an integer data is 4 bytes.

Initializing a one-dimensional array:-

We can initialize an array in two ways.

1. **Static array initialization**

static constant variable that allows you to define a constant value inside a class body.

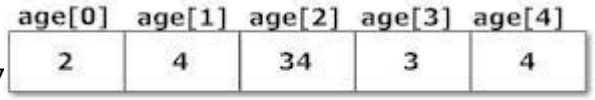
Example

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

{
int age[5]={2,4,34,3,4};
printf("age %d",age[3]);
}
```

Arrays can be initialized at declaration time in this source code as:
int age[5]={2,4,34,3,4};
It is not necessary to define the size of arrays during initialization.
int age[]={2,4,34,3,4};
In this case, the compiler determines the size of array by calculating the number of elements of an array.

2. **Dynamic array initialization**



Initialization of one-dimensional array

A dynamic array lets you keep the number of elements in the array unspecified at the declaration time. You can define the number of elements it holds during run time. Moreover, once declared, the number of elements can be altered at a later point of time too.

Example

```
#include <stdio.h>
int main()
{
   int marks[10],i,n,sum=0;
   printf("Enter number of students: ");
   scanf("%d",&n);
```

```
 for(i=0;i<n;++i)
 {
     printf("Enter marks of student%d: ",i+1);
     scanf("%d",&marks[i]);
 }
for(i=0;i<n;++i){

     printf("%d\n",marks[i]);

 }

}
```

Examples of one dimensional array:-

1. **C program to find the sum marks of n students using arrays**

```
#include <stdio.h>
int main(){
   int i,n,sum=0;
   printf("Enter number of students: ");
   scanf("%d",&n);
   int marks[n];
   for(i=0;i<n;++i){
      printf("Enter marks of student%d: ",i+1);
      scanf("%d",&marks[i]);
      sum+=marks[i];
   }
   printf("Sum= %d",sum);
return 0;
}
```

Output

```
Enter number of students: 3
Enter marks of student1: 12
Enter marks of student2: 31
Enter marks of student3: 2
```

sum=45

2. **Source Code to Display Largest Element of an array**

```
#include <stdio.h>
int main(){
   int i,n;
   printf("Enter size of an array ");
   scanf("%d",&n);
   int arr[n];

   for(i=0;i<n;++i) /* Stores number entered by user. */
   {
     printf("Enter Number %d: ",i+1);
     scanf("%d",&arr[i]);
   }
   for(i=1;i<n;++i) /* Loop to store largest number to arr[0] */
   {
     if(arr[0]<arr[i]) /* Change < to > if you want to find smallest element*/
        arr[0]=arr[i];
   }
   printf("Largest element = %d",arr[0]);
   return 0;
}
```

3. **C Program to Sort the Array in an Ascending Order**

```
#include <stdio.h>
void main()
{
   int i, j, a, n, number[30];
   printf("Enter the value of N \n");
   scanf("%d", &n);
   printf("Enter the numbers \n");
   for (i = 0; i < n; ++i)
      scanf("%d", &number[i]);
   for (i = 0; i < n; ++i)
   {
```

```
      for (j = i + 1; j < n; ++j)
      {
        if (number[i] > number[j])
        {
          a = number[i];

          number[i] = number[j];

          number[j] = a;
        }
      }
    }
    printf("The numbers arranged in ascending order are given below \n");
    for (i = 0; i < n; ++i)
      printf("%d\n", number[i]);
  }
```

4. Write a C program to accept two integer arays and find the sum of the corresponding elements of these two arrays.

```
#include<stdio.h>
void main()
{
int i , n;
printf(" Enter the size of Array A and B\n");
scanf("%d" , &n);
int a[n];
int b[n];
int sum[n];
printf(" Enter the elements of Array A\n");
for(i=0 ; i<n ; i++)
{
scanf("%d" , &a[i]);
}
```

```
printf(" Enter the elements of Array B\n");
for(i=0 ; i<n ; i++)
{
scanf("%d" , &b[i]);
}
for(i=0 ; i<n ; i++)
{
sum[i] = a[i] + b[i];
}
printf(" Sum of elements of A and B\n");
for(i=0 ; i<n ; i++)
{
printf("%d\n" , sum[i] );
}
}
```

5. This Program computes the Sum of Even and Odd Numbers stored in an Array of N

```
#include<stdio.h>
void main()
{
int i , even , odd , n;
printf("Enter the size of Array\n");
scanf("%d" , &n);
int num[n];
even = 0;
odd = 0;
printf(" Enter Array elements\n");
for(i=0 ; i<n ; i++)
{
scanf("%d" , &num[i]);
}
for(i=0 ; i<n ; i++)
{
if((num[i] %2) ==0)
```

```
{
even = even+num[i];
}
else
{
odd =odd+num[i];
}
}
printf(" Sum of even Numbers = %d\n" , even);
printf(" Sum of odd Numbers = %d\n" , odd);
}
```

2. Multidimensional Arrays:-

This multi dimensional arrays are classified into two dimensional, three dimensional and so on n-dimensional array. The dimensionality of an array is determined by the number of subscript present in the given array. If there is one subscript, then it is called a one dimensional array. If there are two subscript, it is called a two-dimensional array and so on. The dimensionality is determined by the number of pairs of square brackets placed after the array name

Example

1. array1[] (one dimensional array)
2. array2[][] (Two dimensional array)
3. array3[][][] (Three dimensional array)

Two dimensional array:

Student Tests	Test 1 [0]	Test 2 [1]	Test 3 [2]
[0]	20	21	22
[1]	18	16	20
[2]	23	24	24
[3]	10	12	17
[4]	08	18	21

It is an ordered table of homogeneous elements. It is generally, referred to as a matrix of some rows and some columns. It is also called two subscripted variable.

Syntax

Datatype arrayname [rows][columns];

Example

int marks[5][3];

This first example specifies that the marks is two dimensional array of 5 rows and 3 columns. Row may represent students and the columns represent tests.

The subscript value ranges from 0 to (size-1). Therefore, the subscript value for student varies from 0 to 4(5 students). Similarly, the subscript value for test varies from 0 to 2.

marks[0][0]=20	marks[0][1]=21	marks[0][2]=22
marks[1][0]=18	marks[1][1]=16	marks[1][2]=20
marks[2][0]=23	marks[2][1]=24	marks[2][2]=24
marks[3][0]=10	marks[3][1]=12	marks[3][2]=17
marks[4][0]=08	marks[4][1]=18	marks[4][2]=21

This indicates that for each row (Students number) element, the colums elements(tests) are rapidly processed. This type of arrangement is known as row majoring or row major order. And the column major order is one where for each column, the row elements are rapidly processed.

Initialization of two dimensional array

Like initialization to a one dimensional array elements, the two dimensional array elements can also be initialized at the time of declaration.

Syntax

Datatype arrayname[size1][size2]={e1,e2,e3,..en};

Example

Int matrix[3][3]={1,2,3,4,5,6,7,8,9};

matrix[0][0]=1;	matrix[0][1]=2;	matrix[0][2]=3;
matrix[1][0]=4;	matrix[1][1]=5;	matrix[1][2]=6;
matrix[2][0]=7;	matrix[2][1]=8;	matrix[2][2]=9;

If the number of elements to be assigned are less than the total number of elements that a two dimensional array has contained, then all the remaining elements of an array are assigned to zeros

Points while initializing two dimensional array

1. The number of sets of initial value must be equal to the number of rows in the arrayname.

2. One to one mapping is preserved. That is the first set of initial values is assigned to the first row element and the second set of initial value is assigned to the second row elements and so on

3.If the number of initial values in each initializing set is less than the number of the corresponding row elements, then all the elements of that row are automatically assigned to zeros.

4. If the number of initial value in each initializing set exceeds the number of the corresponding row elements then there will be a compilation error.

Examples of Multidimensional Array In C

1. **Write a C program to find sum of two matrix of order 2*2 using multidimensional arrays where, elements of matrix are entered by user.**

```
#include <stdio.h>
int main(){
  float a[2][2], b[2][2], c[2][2];
  int i,j;
  printf("Enter the elements of 1st matrix\n");
/* Reading two dimensional Array with the help of two for loop. If there was an
array of 'n' dimension, 'n' numbers of loops are needed for inserting data to array.*/
  for(i=0;i<2;++i)
     for(j=0;j<2;++j){
     printf("Enter a%d%d: ",i+1,j+1);
     scanf("%f",&a[i][j]);
     }
  printf("Enter the elements of 2nd matrix\n");
  for(i=0;i<2;++i)
     for(j=0;j<2;++j){
     printf("Enter b%d%d: ",i+1,j+1);
     scanf("%f",&b[i][j]);
     }
  for(i=0;i<2;++i)
     for(j=0;j<2;++j){
/* Writing the elements of multidimensional array using loop. */
     c[i][j]=a[i][j]+b[i][j]; /* Sum of corresponding elements of two arrays. */
     }
```

```
    printf("\nSum Of Matrix:");
    for(i=0;i<2;++i)
      for(j=0;j<2;++j){
      printf("%.1f\t",c[i][j]);
        if(j==1)          /* To display matrix sum in order. */
          printf("\n");
      }
  return 0;
  }
```

Ouput

Enter the elements of 1st matrix

Enter a11: 2;

Enter a12: 0.5;

Enter a21: -1.1;

Enter a22: 2;

Enter the elements of 2nd matrix

Enter b11: 0.2;

Enter b12: 0;

Enter b21: 0.23;

Enter b22: 23;

Sum Of Matrix:

2.2 0.5

-0.9 25.0

2. C Program to Add Two Matrix in C programming

```
#include <stdio.h>
int main(){
   int r,c,a[100][100],b[100][100],sum[100][100],i,j;
   printf("Enter number of rows (between 1 and 100): ");
   scanf("%d",&r);
   printf("Enter number of columns (between 1 and 100): ");
   scanf("%d",&c);
   printf("\nEnter elements of 1st matrix:\n");
```

```
/* Storing elements of first matrix entered by user. */

   for(i=0;i<r;++i)
     for(j=0;j<c;++j)
     {
        printf("Enter element a%d%d: ",i+1,j+1);
        scanf("%d",&a[i][j]);
     }

/* Storing elements of second matrix entered by user. */

   printf("Enter elements of 2nd matrix:\n");
   for(i=0;i<r;++i)
     for(j=0;j<c;++j)
     {
        printf("Enter element a%d%d: ",i+1,j+1);
        scanf("%d",&b[i][j]);
     }

/*Adding Two matrices */

  for(i=0;i<r;++i)
     for(j=0;j<c;++j)
        sum[i][j]=a[i][j]+b[i][j];

/* Displaying the resultant sum matrix. */
   printf("\nSum of two matrix is: \n\n");
   for(i=0;i<r;++i)
     for(j=0;j<c;++j)
     {
        printf("%d ",sum[i][j]);
        if(j==c-1)
           printf("\n\n");
     }
```

```
    return 0;
}
```

Output:

```
Enter element a12: -4
Enter element a21: 8
Enter element a22: 5
Enter element a31: 1
Enter element a32: 0
Enter elements of 2nd matrix:
Enter element a11: 4
Enter element a12: -7
Enter element a21: 9
Enter element a22: 1
Enter element a31: 4
Enter element a32: 5

Sum of two matrix is:

8  -11
17   6
5    5
```

3. C Program to multiply to matrix in C programming

```
#include <stdio.h>
int main()
{
   int a[10][10], b[10][10], mult[10][10], r1, c1, r2, c2, i, j, k;
   printf("Enter rows and column for first matrix: ");
   scanf("%d%d", &r1, &c1);
   printf("Enter rows and column for second matrix: ");
   scanf("%d%d",&r2, &c2);

/* If colum of first matrix in not equal to row of second matrix, asking user to enter
the size of matrix again. */
```

```
  while (c1!=r2)
  {
     printf("Error! column of first matrix not equal to row of second.\n\n");
     printf("Enter rows and column for first matrix: ");
     scanf("%d%d", &r1, &c1);
     printf("Enter rows and column for second matrix: ");
     scanf("%d%d",&r2, &c2);
  }

/* Storing elements of first matrix. */
  printf("\nEnter elements of matrix 1:\n");
  for(i=0; i<r1; ++i)
  for(j=0; j<c1; ++j)
  {
     printf("Enter elements a%d%d: ",i+1,j+1);
     scanf("%d",&a[i][j]);
  }

/* Storing elements of second matrix. */
  printf("\nEnter elements of matrix 2:\n");
  for(i=0; i<r2; ++i)
  for(j=0; j<c2; ++j)
  {
     printf("Enter elements b%d%d: ",i+1,j+1);
     scanf("%d",&b[i][j]);
  }

/* Initializing elements of matrix mult to 0.*/
  for(i=0; i<r1; ++i)
  for(j=0; j<c2; ++j)
  {
    mult[i][j]=0;
  }
```

```
/* Multiplying matrix a and b and storing in array mult. */
  for(i=0; i<r1; ++i)
  for(j=0; j<c2; ++j)
  for(k=0; k<c1; ++k)
  {
    mult[i][j]+=a[i][k]*b[k][j];
  }

/* Displaying the multiplication of two matrix. */
  printf("\nOutput Matrix:\n");
  for(i=0; i<r1; ++i)
  for(j=0; j<c2; ++j)
  {
    printf("%d ",mult[i][j]);
    if(j==c2-1)
      printf("\n\n");
  }
  return 0;
}
```

Output

```
Enter rows and column for first matrix: 3
2
Enter rows and column for second matrix: 3
2
Error! column of first matrix not equal to row of second.

Enter rows and column for first matrix: 2
3
Enter rows and column for second matrix: 3
2

Enter elements of matrix 1:
Enter elements a11: 3
Enter elements a12: -2
```

Enter elements a13: 5

Enter elements a21: 3

Enter elements a22: 0

Enter elements a23: 4

Enter elements of matrix 2:

Enter elements b11: 2

Enter elements b12: 3

Enter elements b21: -9

Enter elements b22: 0

Enter elements b31: 0

Enter elements b32: 4

Output Matrix:

24 29

6 25

Designing structured programming

If the length of a program is too big, it is very difficult for the programmer to handle it. Normally, large programs are more prone to error and it would be a tedious job to locate and correct the error. Therefore, such programs should be broken down into a number of smaller logical components, each of which serves a specific task.

The process of splitting the lengthy and complex programs into a number of smaller units is called modularization. Programming with such an approach is called modular programming.

Function Definition

A function is a block of code that performs a specific task. Suppose you are defining a function which computes the square of a given number. Then we have to write a set of instructions to do this job.

Types of functions in C programming

Depending on whether a function is defined by the user or already included in C compilers, there are two types of functions in C programming

There are two types of functions in C programming:

- Standard library functions
- User defined functions

Standard library functions

The standard library functions are in-built functions in C programming to handle tasks such as mathematical computations, I/O processing, string handling etc.

These functions are defined in the header file. When you include the header file, these functions are available for use. For example:

The printf() is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in "stdio.h" header file.

There are other numerous library functions defined under "stdio.h", such as scanf(), printf(), getchar() etc. Once you include "stdio.h" in your program, all these functions are available for use.

User-defined functions

As mentioned earlier, C language allows programmer to define functions. Such functions created by the user are called user-defined functions.

Depending upon the complexity and requirement of the program, you can create as many user-defined functions as you want.

Advantages of user-defined function

1. The program will be easier to understand, maintain and debug.
2. Reusable codes that can be used in other programs
3. A large program can be divided into smaller modules. Hence, a large project can be divided among many programmers.

How user-defined function works?

The execution of a C program begins from the main() function. When the compiler encounters functionName(); inside the main function, control of the program jumps to void functionName() And, the compiler starts executing the codes inside the user-defined function. The control of the program jumps to statement next to functionName(); once all the codes inside the function definition are executed. Remember, function name is an identifier and should be unique.

Example

Calculating Factorial

```
#include<stdio.h>
void main()
{
   int n, f;
   int fact(int x); /* Function Prototype declaration */
```

```
   printf("Enter value of n\n");
   scanf("%d",&n);
   f=fact(n);/* calling with single arg here n is actual arg */
   printf("Factorial value = %d\n",f);
}

/* Function Definition (called function) */

int fact(int x)
{
   int i,r=1;
   for(i=1;i<=x;i++)
     r=r*i;
   return(r); // return statement

}
```

Function prototype

A function prototype is simply the declaration of a function that specifies function's name, parameters and return type. It doesn't contain function body.

A function prototype gives information to the compiler that the function may later be used in the program.

Syntax of function prototype

returnType functionName(type1 argument1, type2 argument2,...);

In the above example, int addNumbers(int a, int b); is the function prototype which provides following information to the compiler:

1. name of the function is add()
2. return type of the function is int
3. two arguments of type int are passed to the function

The function prototype is not needed if the user-defined function is defined before the main() function.

Calling a function

Control of the program is transferred to the user-defined function by calling it.

Syntax of function call

functionName(argument1, argument2, ...);

In the above example, function call is made using fact(n);statement from inside the main().

Function definition

Function definition contains the block of code to perform a specific task.

Syntax of function definition

```
returnType functionName(type1 argument1, type2 argument2, ...)
{
  //body of the function
}
```

When a function is called, the control of the program is transferred to the function definition. And, the compiler starts executing the codes inside the body of a function.

Passing arguments to a function

Arguments and parameters are the variable used in a program and a function. Variable used in the function cal are called arguments. These are written within the parentheses followed by the name of the function. Variables used in the function definition are called parameters.

In programming, argument refers to the variable passed to the function. In the above example, one variable n is passed during function call.

The parameter x accepts the passed arguments in the function definition. These argument is called formal parameters of the function.

The type of arguments passed to a function and the formal parameters must match, otherwise the compiler throws error.

If ***n*** is of char type, ***x*** also should be of char type. If ***n*** is of float type, variable ***b*** also should be of float type.

Return statement

The return statement terminates the execution of a function and returns a value to the calling function. The program control is transferred to the calling function after return statement.

In the above example, the value of variable ***r*** is returned to the variable **f** in the main() function.

Syntax of return statement

```
return (expression);
```

user-defined functions can be categorized as:

- Function with no arguments and no return value
- Function with no arguments and a return value

- Function with arguments and no return value
- Function with arguments and a return value.

Example #1: No arguments passed and no return Value

```
#include<stdio.h>
void main()
{
   fact();/* No arguments */
 }

int fact()
{
   int i,r=1,x;
  printf("Enter a number");
   scanf("%d",&x);
   for(i=1;i<=x;i++)
   {
     r=r*i;
   }
   printf("Factorial of a given number is %d",r); // No return statement
}
```

The empty parentheses in fact(); statement inside the main() function indicates that no argument is passed to the function

The return type of the function is void. Hence, no value is returned from the function.

The fact() function takes input from the user, Calculating the factorial of a given number and displays it on the screen.

Example #2: No arguments passed but a return value

```
#include<stdio.h>
main()
{
  int f;
   f=fact();/* No arguments */
   printf("Factorial of a given number is %d",f);
```

```
}

int fact()
{
   int i,r=1,x;
  printf("Enter a number");
   scanf("%d",&x);
   for(i=1;i<=x;i++)
   {
     r=r*i;
   }
   return(r); // return statement
}
```

The empty parentheses in f = fact(); statement indicates that no argument is passed to the function. And, the value returned from the function is assigned to **f.**

Example #3: Argument passed but no return value

```
#include<stdio.h>
main()
{
  int f,n;
  printf("Enter a number");
  scanf("%d",&n);
  f=fact(n);//Passing arguments
 }
fact(int x)
{
   int i,r=1;
   for(i=1;i<=x;i++)
   {
     r=r*i;
   }
  printf("Factorial of a given number is %d",r)//No retrun statemet;
}
```

The integer value entered by the user is passed to fact() function.
Here, the fact() function calculate the factorial of a given number and displays the appropriate message.

Example #4: Argument passed and a return value

```
#include<stdio.h>
void main()
{
   int n, f;
   int fact(int x); /* Function Prototype declaration */

   printf("Enter value of n\n");
   scanf("%d",&n);
   f=fact(n);/* calling with single arg here n is actual arg */
   printf("Factorial value = %d\n",f);
}

/* Function Definition (called function) */

int fact(int x)
{
   int i,r=1;
   for(i=1;i<=x;i++)
     r=r*i;
   return(r); // return statement
}
```

The input from the user is passed to fact() function.
The fact() function calculate the factorial of a given number.The return value is assigned to variable f.
Then, the appropriate message is displayed from the main() function.

Inter function communication

In the above programs arguments are passed to the functions and their values are copied in the corresponding function. This is sort of information interchange between a calling function and a called function. This process of transmitting the values from one function to other is known as parameter passing. There are two methods of parameters passing.

1. Call by value
2. Call by reference

1. Call by value

When the values of the arguments are passed from a calling function to a called function, the values are copied into the called function. If any changes are made to the values in the called function, there will not be any change in the original values within the calling function.

Example

```
#include<stdio.h>
void interchange(int number1,int number2)
{
   int temp;
   temp = number1;
   number1 = number2;
   number2 = temp;
}
int main() {
   int num1=50,num2=70;
   interchange(num1,num2);
   printf("\nNumber 1 : %d",num1);
   printf("\nNumber 2 : %d",num2);
   return(0);
}
```

Output :

Number 1 : 50

Number 2 : 70

Explanation

1. While Passing Parameters using call by value , **xerox copy of original parameter is created** and passed to the called function.
2. Any update made inside method will not affect the **original value of variable in calling**

function.

3. In the above example num1 and num2 are the original values and xerox copy of these values is passed to the function and these values are copied into number1,number2 variable of sum function respectively.
4. As their scope is limited to only function so they **cannot alter the values inside main function**.

2. Call by reference

In this method, the actual values are not passed, instead their addresses are passed to a calling function. Here, no values are copied as the memory locations themselves are referenced. If any modification is made to the values in the called function, then the original values will get changed within the calling function.

Example

```
#include<stdio.h>

void interchange(int *num1,int *num2)
{
   int temp;
   temp = *num1;
   *num1 = *num2;
   *num2 = temp;
}
int main() {
   int num1=50,num2=70;
   interchange(&num1,&num2);
   printf("\nNumber 1 : %d",num1);
   printf("\nNumber 2 : %d",num2);
   return(0);
}
```

Output :

Number 1 : 70

Number 2 : 50

Explanation

1. While passing parameter using call by address scheme , we are **passing the actual address**

of the variable to the called function.

2. Any updates made inside the called function **will modify the original copy** since we are directly modifying the content of the exact memory location.

C Standard Library Functions

C Standard library functions or simply C Library functions are inbuilt functions in C programming. Function prototype and data definitions of these functions are written in their respective header file. For example: If you want to use printf() function, the header file <stdio.h> should be included.

```
#include <stdio.h>
int main()
  {
/* If you write printf() statement without including header file, this program will show error. */
     printf("Catch me if you can.");
  }
```

There is at least one function in any C program, i.e., the **main()** function (which is also a library function). This program is called at program starts.

There are many library functions available in C programming to help the programmer to write a good efficient program.

Suppose, you want to find the square root of a number. You can write your own piece of code to find square root but, this process is time consuming and the code you have written may not be the most efficient process to find square root. But, in C programming you can find the square root by just using **sqrt()** function which is defined under header file **"math.h"**

Use Of Library Function To Find Square root

```
#include <stdio.h>
#include <math.h>
int main(){
  float num,root;
  printf("Enter a number to find square root.");
  scanf("%f",&num);
  root=sqrt(num);        /* Computes the square root of num and stores in root. */
  printf("Square root of %.2f=%.2f",num,root);
  return 0;
}
```

List of Standard Library Functions Under Different Header Files in C Programming

1. **stdio.h: I/O functions:**
    a. **getchar()**returns the next character typed on the keyboard.
    b. **putchar()**outputs a single character to the screen.
    c. **printf()**to print the statement
    d. **scanf()**to read the value from key board
2. **string.h: String functions**
    a. **strcat()**concatenates a copy of str2 to str1
    b. **strcmp()**compares two strings
    c. **strcpy()**copys contents of str2 to str1

ctype.h: Character functions

a. **isdigit()**returns non-0 if arg is digit 0 to 9
b. **isalpha()**returns non-0 if arg is a letter of the alphabet
c. **isalnum()**returns non-0 if arg is a letter or digit
d. **islower()**returns non-0 if arg is lowercase letter
e. **isupper()**returns non-0 if arg is uppercase letter

math.h: Mathematics functions

a. **acos()**returns arc cosine of arg
b. **asin()**returns arc sine of arg
c. **atan()**returns arc tangent of arg
d. **cos()**returns cosine of arg
e. **exp()**returns natural logarithim e
f. **fabs()**returns absolute value of num
g. **sqrt()**returns square root of num

time.h: Time and Date functions

a. **time()**returns current calender time of system
b. **difftime()**returns difference in secs between two times
c. **clock()**returns number of system clock cycles since program execution

stdlib.h:Miscellaneous functions

a. **malloc()**provides dynamic memory allocation, covered in future sections
b. **rand()**as already described previously
c. **srand()**used to set the starting point for rand()

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.

Storage Classes

There are four storage classes in c. These storage classes are used to define the scope and the life-time of the variable and/or functions.

1. automatic or local variables
2. external or global variables
3. static Variables
4. register Variables

Automatic or Local Variable

All variables declared are of type Auto by default.The local variables exist only inside the function in which it is declared. When the function exits, the local variables are destroyed.In order to Explicit declaration of variable use 'auto' keyword.

Example

```
#include<stdio.h>
int i=10; //Global declaration
void main()
{
int j;
auto int k=50;//local variable
printf ("i=%d",i);
j=value(k);
printf("j=%d",j);
}
//function to compute values
int value(int a)
{
int k;
k=a+10;
return(k);
```

```
//End of the function
}
```

Output

```
i=10
j=60
```

External or Global Variable

Variables that are declared outside of all functions are known as external variables. External or global variables are accessible to any function.Generally , External variables are declared again in the function using keyword extern

Example1:

```
#include<stdio.h>
int num = 75 ;
void display();
void main()
{
 //extern int num ;
 printf("nNum : %d",num);
 display();
}
void display()
{
 //extern int num ;
  printf("nNum : %d",num);
}
```

1. Declaration within the function indicates that the function uses external variable
2. Functions belonging to same source code , does not require declaration (no need to write extern)
3. If variable is defined outside the source code , then declaration using extern keyword is required

Example 2:

```
#include<stdio.h>
int i=10; //Global declaration
void main()
```

```
{
int j;
printf ("i=%d",i);
j=value();
printf("j=%d",j);
}
//function to compute values
int value()
{
int k;
k=i+10;//i is not declared in the function because it is a global variable
return(k);
//End of the function
}
```

Output

```
i=10
j=20
```

Register Variable

1. register keyword is used to define local variable.
2. Local variable are stored in register instead of **RAM**.
3. As variable is stored in register, the **Maximum size of variable = Maximum Size of Register**
4. unary operator [&] is not associated with it because Value is not stored in RAM instead it is stored in Register.
5. This is generally used for **faster access**.
6. Common use is "**Counter**"
7. It is not applicable for arrays, structures or pointers.

Syntax

```
{
register int count;
}
```

Example

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

```
{
int num1,num2;
register int sum;
printf("\nEnter the Number 1 : ");
scanf("%d",&num1);
printf("\nEnter the Number 2 : ");
scanf("%d",&num2);
sum = num1 + num2;
printf("\nSum of Numbers : %d",sum);
return(0);
}
```

Explanation of program

1. In the above program we have declared two variables num1,num2. These two variables are stored in RAM.
2. Another variable is declared which is stored in register variable.Register variables are stored in the register of the microprocessor.Thus memory access will be faster than other variables.
3. If we try to declare more register variables then it can treat variables as Auto storage variables as memory of microprocessor is fixed and limited.

Static Variable

Static variables have a property of preserving their value even after they are out of their scope! Hence, static variables preserve their previous value in their previous scope and are not initialized again in the new scope .A static int variable remains in memory while the program is running. A normal or auto variable is destroyed when a function call where the variable was declared is over.

Syntax:

```
static data_type var_name = var_value;
```

Example 1:Noramal Variable

```
#include<stdio.h>
int fun()
{
  int count = 0;
  count++;
```

```
  return count;
}

int main()
{
  printf("%d ", fun());
  printf("%d ", fun());
  return 0;
//A normal or auto variable is destroyed when a function call where the variable was declared is over.
}
```

Output

```
1 1
```

Example 2: Static Variable

```
#include <stdio.h>
void display();

int main()
{
   display();
   display();
}
void display()
{
   static int c = 0;
   printf("%d ",c);
   c += 5;
//A static int variable remains in memory while the program is running.
}
```

Output

```
0  5
```

During the first function call, the value of *c* is equal to 0. Then, it's value is increased by 5.

During the second function call, variable *c* is not initialized to 0 again. It's because *c* is a static variable. So, 5 is displayed on the screen.

Type Qualifiers

The keywords which are used to modify the properties of a variable are called type qualifiers.There are two types of qualifiers available in C language. They are,

1. constant Keyword
2. volatile Keyword

1. CONST KEYWORD:

Constants are also like normal variables. But, only difference is, their values can't be modified by the program once they are defined.

They refer to fixed values. They are also called as literals.

They may be belonging to any of the data type.

Syntax:

const data_type variable_name; (or) const data_type *variable_name;

Please refer C – Constants topic in this tutorial for more details on const keyword.

2. VOLATILE KEYWORD:

When a variable is defined as volatile, the program may not change the value of the variable explicitly.

But, these variable values might keep on changing without any explicit assignment by the program. These types of qualifiers are called volatile.

For example, if global variable's address is passed to clock routine of the operating system to store the system time, the value in this address keep on changing without any assignment by the program. These variables are named as volatile variable.

Syntax:

volatile data_type variable_name; (or) volatile data_type *variable_name;

Recursion

A function that calls itself is known as recursive function. And, this technique is known as recursion. That is a function which perform a particular task is repeatedly calling itself. There must be an exclusive stopping statement(terminating condition) in a recursive function .Otherwise, the function will not be terminated. It enters into an infinite loop.

Example

```
#include <stdio.h>
long int multiplyNumbers(int n);

int main()
```

```
{
   int n;
   printf("Enter a positive integer: ");
   scanf("%d", &n);
   printf("Factorial of %d = %ld", n, multiplyNumbers(n));
   return 0;
}
long int multiplyNumbers(int n)
{
   if (n >= 1)
      return n*multiplyNumbers(n-1);// multiplyNumbers function calls itself
   else
      return 1;
}
```

Advantages and Disadvantages of Recursion

Recursion makes program elegant and cleaner. All algorithms can be defined recursively which makes it easier to visualize and prove.

If the speed of the program is vital then, you should avoid using recursion. Recursions use more memory and are generally slow.

C -Preprocessors

The **C Preprocessor** is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation. We'll refer to the C Preprocessor as CPP.

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column. The following section lists down all the important preprocessor directives –

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 and #if in one statement.
#endif	Ends preprocessor conditional.

Predefined Macros

ANSI C defines a number of macros. Although each one is available for use in programming, the predefined macros should not be directly modified.

Example

```
#include <stdio.h>
main(){
  printf("File :%s\n", __FILE__);
  printf("Date :%s\n", __DATE__);
  printf("Time :%s\n", __TIME__);
  printf("Line :%d\n", __LINE__);
  printf("ANSI :%d\n", __STDC__);
}
```

Macro	Description
__DATE__	The current date as a character literal in "MMM DD YYYY" format.
__TIME__	The current time as a character literal in "HH:MM:SS" format.
__FILE__	This contains the current filename as a string literal.
__LINE__	This contains the current line number as a decimal constant.

Strings

Definition

- C Strings are nothing but array of characters ended with null character (‘\0’).
- This null character indicates the end of the string.

- Strings are always enclosed by double quotes. Whereas, character is enclosed by single quotes in C.

Declaration of strings

Before we actually work with strings, we need to declare them first. Strings are declared in a similar manner as arrays. Only difference is that, strings are of char type.

Example

```
char s[5];
```

Initialization of strings

In C, string can be initialized in a number of different ways. For convenience and ease, both initialization and declaration are done in the same step.

Example

```
char c[] = "abcd";
   OR,
char c[50] = "abcd";
   OR,
char c[] = {'a', 'b', 'c', 'd', '\0'};
   OR,
char c[5] = {'a', 'b', 'c', 'd', '\0'};
```

Reading Strings from user

You can use the scanf() function to read a string like any other data types. But we don't use the address of operator(&) while passing an array of character(strings) to a scanf() function because the array name itself returns the starting address of the entire array.

However, the scanf() function only takes the first entered word. The function terminates when it encounters a white space (or just space).

Example

```
#include <stdio.h>
int main()
{
   char name[20];
   printf("Enter name: ");
   scanf("%s", name);
   printf("Your name is %s.", name);
   return 0;
```

```
}
```

Output

```
Enter name: Dennis Ritchie
Your name is Dennis.
```

Reading a line of text

An approach to reading a full line of text is to read and store each character one by one.

Using getchar() to read a line of text

using the function getchar(), ch gets a single character from the user each time. This process is repeated until the user enters return (enter key). Finally, the null character is inserted at the end to make it a string.

Example

```
#include <stdio.h>
int main()
{
   char name[30], ch;
   int i = 0;
   printf("Enter name: ");
   while(ch != '\n')    // terminates if user hit enter
   {
      ch = getchar();
      name[i] = ch;
      i++;
   }
   name[i] = '\0';      // inserting null character at end
   printf("Name: %s", name);
   return 0;
}
```

Using standard library function to read a line of text

There are predefined functions gets() and puts in C language to read and display string respectively.

Example

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

```
   char name[30];
   printf("Enter name: ");
   gets(name);     //Function to read string from user.
   printf("Name: ");
   puts(name);    //Function to display string.
   return 0;
}
```

Output

```
Enter name: RGUKT IIIT
Name: RGUKT IIIT
```

Note: This gets() function is dangerous because there is no way for the method to know how much place has been allocated to that char * in ever situation. Instead of gets() function you can use fgets() function, which allows you to limit the number of characters read, so the buffer does not overflow.

Example

```
#include <stdio.h>
int main()
{
   char name[5];
   printf("Enter name: ");
   fgets(name,5,stdin);     //Function to read string from user.
   printf("Name: ");
   puts(name);    //Function to display string.
   return 0;
}
```

Passing Strings to Functions

Strings are just char arrays. So, they can be passed to a function in a similar manner as arrays.

Example

```
#include <stdio.h>
void displayString(char str[]);
int main()
{
   char str[50];
   printf("Enter string: ");
```

```
   gets(str);
   displayString(str);     // Passing string c to function.
   return 0;
}
void displayString(char str[]){
   printf("String Output: ");
   puts(str);
}
```

Example 1

Reverse of a given String

```
#include<stdio.h>
#include<string.h>
int main() {
  char str[100], temp;
  int i, j = 0;
  printf("\nEnter the string :");
  gets(str);
  i = 0;
  j = strlen(str) - 1;
  while (i < j) {
    temp = str[i];
    str[i] = str[j];
    str[j] = temp;
    i++;
    j--;
  }
  printf("\nReverse string is :%s", str);
  return (0);
}
```

Example 2

Write a program to take a sentence as input and reverse every word of the sentence.

```
   #include <stdio.h>
   #include <string.h>
   void main()
```

```
{
   int i, j = 0, k = 0, x, len;
   char str[100], str1[10][20], temp;
   printf("enter the string :");
   scanf("%[^\n]s", str);
/* reads into 2d character array */
   for (i = 0;str[i] != '\0'; i++)
   {
      if (str[i] == ' ')
      {
         str1[k][j]='\0';
         k++;
         j=0;
      }
      else
      {
         str1[k][j]=str[i];
         j++;
      }
   }
   str1[k][j] = '\0';
/* reverses each word of a given string */
   for (i = 0;i <= k;i++)
   {
      len = strlen(str1[i]);
      for (j = 0, x = len - 1;j < x;j++,x--)
      {
         temp = str1[i][j];
         str1[i][j] = str1[i][x];
         str1[i][x] = temp;
      }
   }
   for (i = 0;i <= k;i++)
   {
```

```
            printf("%s ", str1[i]);
        }
    }
```

Example3

Program to count vowels, consonants etc.

```
#include <stdio.h>

int main()
{
    char line[150];
    int i, vowels, consonants, digits, spaces;

    vowels = consonants = digits = spaces = 0;

    printf("Enter a line of string: ");
    scanf("%[^\n]", line);

    for(i=0; line[i]!='\0'; ++i)
    {
        if(line[i]=='a' || line[i]=='e' || line[i]=='i' ||
           line[i]=='o' || line[i]=='u' || line[i]=='A' ||
           line[i]=='E' || line[i]=='I' || line[i]=='O' ||
           line[i]=='U')
        {
            ++vowels;
        }
        else if((line[i]>='a'&& line[i]<='z') || (line[i]>='A'&& line[i]<='Z'))
        {
            ++consonants;
        }
        else if(line[i]>='0' && line[i]<='9')
        {
            ++digits;
        }
```

```
    else if (line[i]==' ')
    {
       ++spaces;
    }
  }

  printf("Vowels: %d",vowels);
  printf("\nConsonants: %d",consonants);
  printf("\nDigits: %d",digits);
  printf("\nWhite spaces: %d", spaces);

  return 0;
}
```

Output

```
Enter a line of string: adfslkj34 34lkj343 34lk
Vowels: 1
Consonants: 11
Digits: 9
White spaces: 2
```

Example4

Remove Characters in String Except Alphabets

```
#include<stdio.h>

int main()
{
  char line[150];
  int i, j;
  printf("Enter a string: ");
  gets(line);

  for(i = 0; line[i] != '\0'; ++i)
  {
    while (!( (line[i] >= 'a' && line[i] <= 'z') || (line[i] >= 'A' && line[i] <= 'Z') || line[i] == '\0') )
    {
```

```
        for(j = i; line[j] != '\0'; ++j)
        {
           line[j] = line[j+1];
        }
        line[j] = '\0';
      }
   }
   printf("Output String: ");
   puts(line);
   return 0;
}
```

Output

```
Enter a string: google@google.com
Output String: googlegooglecom
```

Example 5

Find the Frequency of Characters

```
#include <stdio.h>

int main()
{
  char str[1000], ch;
  int i, frequency = 0;
  printf("Enter a string: ");
  gets(str);
  printf("Enter a character to find the frequency: ");
  scanf("%c",&ch);
  for(i = 0; str[i] != '\0'; ++i)
  {
     if(ch == str[i])
        ++frequency;
  }
  printf("Frequency of %c = %d", ch, frequency);
  return 0;
}
```

Output

Enter a string: This website is awesome.

Enter a character to find the frequency: e

Frequency of e = 4

C String functions:

String.h header file supports all the string functions in C language. All the string functions are given below.

Strcat ( )

strcat( ) function in C language concatenates two given strings. It concatenates source string at the end of destination string. Syntax for strcat( ) function is given below.

Example

```
#include <stdio.h>
#include <string.h>
int main( )
{
  char source[ ] = " Hello RGUKT" ;
  char target[ ]= " Students" ;
  printf ( "\nSource string = %s", source ) ;
  printf ( "\nTarget string = %s", target ) ;
  strcat ( target, source ) ;
  printf ( "\nTarget string after strcat( ) = %s", target ) ;
}
```

Output

Target string after strcat( ) = Students Hello RGUKT

C - strncat()

strncat( ) function in C language concatenates (appends) portion of one string at the end of another string.

Example

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

```
#include <string.h>

int main( )
{
  char source[ ] = " Hello RGUKT" ;
  char target[ ]= " Students" ;

  printf ( "\nSource string = %s", source ) ;
  printf ( "\nTarget string = %s", target ) ;

  strncat ( target, source,9) ;

  printf ( "\nTarget string after strcat( ) = %s", target ) ;
}
```

Output

```
Target string after strcat( ) = Students Hello R
```

C - strcpy() function

strcpy( ) function copies contents of one string into another string. Syntax for strcpy function is given below.

```
char * strcpy ( char * destination, const char * source );
```

- Example:
  strcpy (str1, str2) – It copies contents of str2 into str1.
  strcpy (str2, str1) – It copies contents of str1 into str2.
- If destination string length is less than source string, entire source string value won't be copied into destination string.
- For example, consider destination string length is 20 and source string length is 30. Then, only 20 characters from source string will be copied into destination string and remaining 10 characters won't be copied and will be truncated.

Example

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

```
int main( )
{
  char source[ ] = "Hello students" ;
  char target[]= "" ;
  printf ( "\nsource string = %s", source ) ;
  printf ( "\ntarget string = %s", target ) ;
  strcpy ( target, source ) ;
  printf ( "\ntarget string after strcpy( ) = %s", target ) ;
  return 0;
}
```

Output

```
source string = Hello students
target string =
target string after strcpy( ) = Hello students
```

C - strncpy() function

strncpy( ) function copies portion of contents of one string into another string.

Example

```
#include <stdio.h>
#include <string.h>
int main( )
{
  char source[ ] = "Hello Students" ;
  char target[20]= "" ;
  printf ( "\nsource string = %s", source ) ;
printf ( "\ntarget string = %s", target ) ;
  strncpy ( target, source, 5 ) ;
  printf ( "\ntarget string after strcpy( ) = %s", target ) ;
return 0;
}
```

Output

source string = Hello Students

target string =

target string after strcpy( ) = Hello

C - strlen() function

strlen( ) function in C gives the length of the given string.

- strlen( ) function counts the number of characters in a given string and returns the integer value.
- It stops counting the character when null character is found. Because, null character indicates the end of the string in C.

Example

```
#include <stdio.h>
#include <string.h>
int main( )
{
   int len;
   char array[20]="RGUKT" ;
   len = strlen(array) ;
   printf ( "string length = %d \n" , len ) ;
   return 0;
}
```

Output

string length = 5

C - strchr() function

strchr( ) function returns pointer to the first occurrence of the character in a given string.

Example

```
#include <stdio.h>
#include <string.h>
int main ()
{
 char string[55] ="This is a string for testing";
 char *p;
 p = strchr (string,'i');

 printf ("Character i is found at position %d\n",p-string+1);
 printf ("First occurrence of character \"i\" in \"%s\" is" \
        " \"%s\"",string, p);

  return 0;
}
```

Output

Character i is found at position 3

First occurrence of character “i” in “This is a string for testing” is “is is a string for testing”

C - strrchr() function

strrchr( ) function in C returns pointer to the last occurrence of the character in a given string.

Example

```
#include <stdio.h>
#include <string.h>
int main ()
{
 char string[55] ="This is a string for testing";
 char *p;
 p = strrchr (string,'i');
```

```
  printf ("Character i is found at position %d\n",p-string+1);
  printf ("Last occurrence of character \"i\" in \"%s\" is" \
          " \"%s\"",string, p);

   return 0;
}
```

Output

Character i is found at position 26

Last occurrence of character “i” in “This is a string for testing” is “ing”

C - strstr() function

strstr( ) function returns pointer to the first occurrence of the string in a given string.

Example

```
#include <stdio.h>
#include <string.h>
int main ()
{
 char string[55] ="This is a test string for testing";
 char *p;
 p = strstr (string,"test");
 if(p)
 {
  printf("string found\n" );
  printf ("First occurrence of string \"test\" in \"%s\" is"\
       " \"%s\"",string, p);
 }
 else printf("string not found\n" );
  return 0;
}
```

Output

string found

First occurrence of string “test” in “This is a test string for testing” is “test string for testing”

C - strrstr() function

strrstr( ) function returns pointer to the last occurrence of the string in a given string.

Example

```
#include <stdio.h>
#include <string.h>
int main ()
{
 char string[55] ="This is a test string for testing";
 char *p;
 p = strrstr (string,"test");
 if(p)
```

```
 {
   printf("string found\n" );
   printf ("Last occurrence of string \"test\" in \"%s\" is"\
         " \"%s\"",string, p);
 }
 else printf("string not found\n" );
  return 0;
}
```

Output

string found

Last occurrence of string “test” in “This is a test string for testing” is “testing”

C - strlwr() function

strlwr( ) function converts a given string into lowercase.

Example

```
#include<stdio.h>
#include<string.h>
int main()
{
   char str[ ] = "MODIFY This String To LOwer";
   printf("%s\n",strlwr (str));
   return  0;
}
```

Output

modify this string to lower

C - strupr() function

strupr( ) function converts a given string into uppercase.

Example

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

int main()
{
```

```
   char str[ ] = "Modify This String To Upper";
   printf("%s\n",strupr(str));
   return  0;
}
```

Output:

MODIFY THIS STRING TO UPPER

C - strtok() function

strtok( ) function in C tokenizes/parses the given string using delimiter

Example

```
#include <stdio.h>
#include <string.h>
int main ()
{
 char string[50] ="Test,string1,Test,string2:Test:string3";
 char *p;
 printf ("String \"%s\" is split into tokens:\n",string);
 p = strtok (string,",:");
 while (p!= NULL)
 {
  printf ("%s\n",p);
  p = strtok (NULL, ",:");
 }
 return 0;
}
```

Output

String “Test,string1,Test,string2:Test:string3” is split into tokens:

```
Test
string1
Test
string2
Test
string3
```

C - strset() function

strset( ) function sets all the characters in a string to given character.

Example

```
#include<stdio.h>
#include<string.h>
int main()
{
  char str[20] = "Test String";
  printf("Original string is : %s", str);
```

```
  printf("Test string after strset() : %s",strset(str,'#'));
  printf("After string set: %s",str);
  return 0;
}
```

Output

```
Original string is          : Test String
Test string after strset() : ###########
```

C - strnset() function

strnset( ) function sets portion of characters in a string to given character

Example

```
#include<stdio.h>
#include<string.h>
int main()
{
   char str[20] = "Test String";
   printf("Original string is : %s", str);
   printf("Test string after string n set" \
        " : %s", strnset(str,'#',4));
   printf("After string n set : %s", str);
   return 0;
}
```

Output

```
Original string is              : Test String
Test string after string set : #### String
```

C - strrev() function

strrev( ) function reverses a given string in C language.

Example

```
#include<stdio.h>
#include<string.h>
int main()
{
  char name[30] = "Hello";
```

```
  printf("String before strrev( ) : %s\n",name);
  printf("String after strrev( ) : %s",strrev(name));
  return 0;
}
```

Output

String before strrev( ) : Hello

String after strrev( ) : olleH

POINTERS

Introduction to Pointers

Pointers are variables that hold address of another variable of same data type. Pointers are one of the most distinct and exciting features of C language. It provides power and flexibility to the language.

Benefit of using pointers

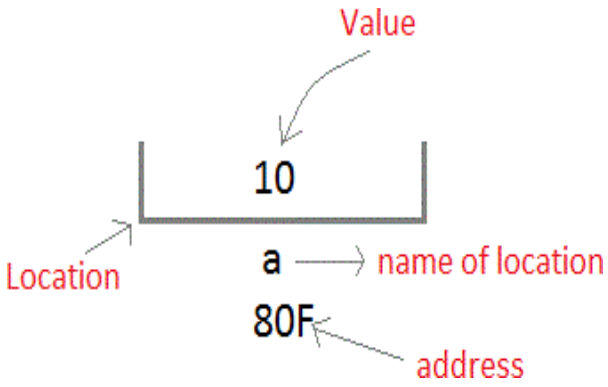
- Pointers provide direct access to memory
- Pointers provide a way to return more than one value to the functions
- Reduces the storage space and complexity of the program
- Reduces the execution time of the program
- Provides an alternate way to access array elements
- Pointers can be used to pass information back and forth between the calling function and called function.
- Pointers allow us to perform dynamic memory allocation and deallocation.
- Pointers helps us to build complex data structures like linked list, stack, queues, trees, graphs etc.
- Pointers allows us to resize the dynamically allocated memory block.
- Addresses of objects can be extracted using pointers

Concept of Pointer

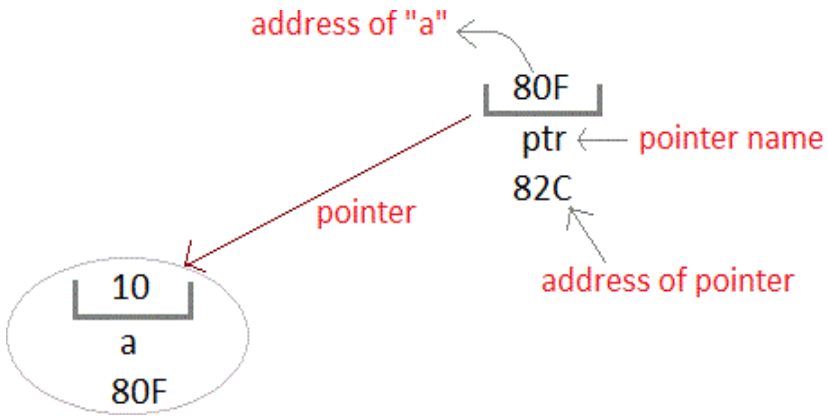
Whenever a **variable** is declared, system will allocate a location to that variable in the memory, to hold value. This location will have its own address number.

Let us assume that system has allocated memory location 80F for a variable **a**.

Example:- int a = 10 ;



We can access the value 10 by either using the variable name **a** or the address 80F. Since the memory addresses are simply numbers they can be assigned to some other variable. The variable that holds memory address are called **pointer variables**. A **pointer** variable is therefore nothing but a variable that contains an address, which is a location of another variable. Value of **pointer variable** will be stored in another memory location.



Pointer variables

In C, there is a special variable that stores just the address of another variable. It is called Pointer variable or, simply, a pointer.

Declaration of Pointer

Syntax:- data_type* pointer_variable_name;

Example:- int* p;

Key points to remember about pointers in C:

- Normal variable stores the value whereas pointer variable stores the address of the variable.
- The content of the C pointer always be a whole number i.e. address.
- Always C pointer is initialized to null, i.e. int *p = null.
- The value of null pointer is 0.
- & symbol is used to get the address of the variable.
- * symbol is used to get the value of the variable that the pointer is pointing to.
- If pointer is assigned to NULL, it means it is pointing to nothing.
- Two pointers can be subtracted to know how many elements are available between these two pointers.
- But, Pointer addition, multiplication, division are not allowed.
- The size of any pointer is 4 byte (for 32 bit compiler).
-

Reference operator (&) and Dereference operator (*)

The & is called reference operator. It gives you the address of a variable. Likewise, there is another operator that gets you the value from the address, it is called a dereference operator (*).

Example

```
/* Source code to demonstrate, handling of pointers in C program */
#include <stdio.h>
int main(){
  int* pc;
  int c;
  c=22;
  printf("Address of c:%u\n",&c);
  printf("Value of c:%d\n\n",c);

  pc=&c;
  printf("Address of pointer pc:%u\n",pc);
  printf("Content of pointer pc:%d\n\n",*pc);
  c=11;
  printf("Address of pointer pc:%u\n",pc);
  printf("Content of pointer pc:%d\n\n",*pc);
  *pc=2;
  printf("Address of c:%u\n",&c);
```

```
    printf("Value of c:%d\n\n",c);
    return 0;
}
```

Output

Address of c: 2686784

Value of c: 22

Address of pointer pc: 2686784

Content of pointer pc: 22

Address of pointer pc: 2686784

Content of pointer pc: 11

Address of c: 2686784

Value of c: 2

Pointers to Pointers

A pointer to a pointer is a form of multiple indirection, 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.

Declaration of pointer to pointer

Example:- 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>
int 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 available at *ptr = %d\n", *ptr );
  printf("Value available at **pptr = %d\n", **pptr);
  return 0;
}
```

Output

```
Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000
```

Compatibility

The rules for assigning one pointer to another are tighter than the rules for numeric types. For example, you can assign an int value to a double variable without using a type conversion, but you can't do the same for pointers to these two types

Example

```
/*
 * ptr_compatibility.c -- program illustrates concept of pointer
 * compatibility
 */
#include <stdio.h>

int main(void)
```

```
{
   int n = 5;
   long double x;

   int *pi = &n;
   long double *pld = &x;

   x = n;     /* implicit type conversion */
   pld = pi; /* compile-time error: assigning pointer-to-int to */
            /* pointer-to-long-double */

   return 0;
}
```

Void Pointers

- Suppose we have to declare integer pointer,character pointer and float pointer then we need to declare 3 pointer variables.
- Instead of declaring different types of pointer variable it is feasible to declare single pointer variable which can act as integer pointer,character pointer.
- **Advantage of void pointers is** this allows these functions to be used to allocate memory of any data type.
- **Disadvantage of void pointers is** void pointers cannot be dereferenced

Example

```
int main()
{
   int a = 10;
   void *ptr = &a; // void pointer holds address of int 'a'
   printf("%d", *ptr);
   return 0;
}
```

Arrays and Pointers

Arrays are closely related to pointers in C programming but the important difference between them is that, a pointer variable takes different addresses as value whereas, in case of

array it is fixed. There may be a situation when we want to maintain an array, which can store pointers to an int or char or any other data type available.

Example

```
#include <stdio.h>
int main()
{
  char charArr[4];
  int i;

  for(i = 0; i < 4; ++i)
  {
    printf("Address of charArr[%d] = %u\n", i, &charArr[i]);
  }

  return 0;
}
```

Output

Address of charArr[0] = 28ff44

Address of charArr[1] = 28ff45

Address of charArr[2] = 28ff46

Address of charArr[3] = 28ff47

Relation between Arrays and Pointers

int arr[4];

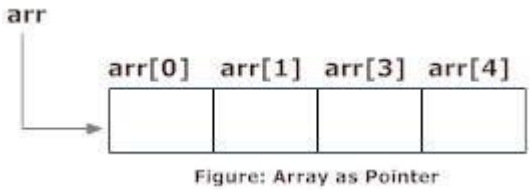


Figure: Array as Pointer

The name of the array always points to address of the first element of an array. In the above example, *arr* and *&arr[0]* points to the address of the first element.

&arr[0] is equivalent to arr

Since, the addresses of both are the same, the values of arr and &arr[0] are also the same.

arr[0] is equivalent to *arr (value of an address of the pointer)

Similarly,

&arr[1] is equivalent to (arr + 1) AND, arr[1] is equivalent to *(arr + 1).

&arr[2] is equivalent to (arr + 2) AND, arr[2] is equivalent to *(arr + 2).

&arr[3] is equivalent to (arr + 1) AND, arr[3] is equivalent to *(arr + 3).

.

.

&arr[i] is equivalent to (arr + i) AND, arr[i] is equivalent to *(arr + i).

In C, you can declare an array and can use pointer to alter the data of an array.

Example

```
#include <stdio.h>
int main()
{
 int i, classes[6],sum = 0;
 printf("Enter 6 numbers:\n");
 for(i = 0; i < 6; ++i)
 {
    // (classes + i) is equivalent to &classes[i]
    scanf("%d",(classes + i));
    // *(classes + i) is equivalent to classes[i]
    sum += *(classes + i);
 }
 printf("Sum = %d", sum);
 return 0;

}
```

Output

Enter 6 numbers:

2

3

4

5

3

4

Sum = 21

Constant Pointers

A constant pointer is a pointer that cannot change the address its holding. In other words, we can say that once a constant pointer points to a variable then it cannot point to any other variable.

Syntax:- <type of pointer> * const <name of pointer>

Example:- int * const ptr;

Example:-

```
#include<stdio.h>
int main(void)
{
   int var1 = 0, var2 = 0;
   int *const ptr = &var1;
   ptr = &var2;
   printf("%d\n", *ptr);
   return 0;
}
```

In the above example :

- We declared two variables var1 and var2
- A constant pointer 'ptr' was declared and made to point var1
- Next, ptr is made to point var2.
- Finally, we try to print the value ptr is pointing to.

So, in a nutshell, we assigned an address to a constant pointer and then tried to change the address by assigning the address of some other variable to the same constant pointer.

Pointers and Strings

A pointer which pointing to an array which content is string, is known as pointer to array of strings.

Syntax:- char *p;

Initialization of strings Using pointers

```
char *c = "abcd";
```

Examples

1 : Printing Address of the Character Array

```
#include<stdio.h>
int main()
{
int i;
char *arr[4] = {"C","C++","Java","VBA"};
char *(*ptr)[4] = &arr;
for(i=0;i<4;i++)
   printf("Address of String %d : %u\n",i+1,(*ptr)[i]);

return 0;
}
```

Output :

Address of String 1 = 178

Address of String 2 = 180

Address of String 3 = 184

Address of String 4 = 189

2. Printing Contents of character array

```
#include<stdio.h>
int main()
{
int i;
char *arr[4] = {"C","C++","Java","VBA"};
char *(*ptr)[4] = &arr;
for(i=0;i<4;i++)
   printf("String %d : %s\n",i+1,(*ptr)[i]);
return 0;
}
```

Output :

String 1 = C

String 2 = C++

String 3 = Java

String 4 = VBA

Pointers to Functions

When a pointer is passed as an argument to a function, address of the memory location is passed instead of the value. This is because, pointer stores the location of the memory, and not the value.

Example

/* C Program to swap two numbers using pointers and function. */

```
#include <stdio.h>
void swap(int *n1, int *n2);
int main()
{
   int num1 = 5, num2 = 10;
   // address of num1 and num2 is passed to the swap function
   swap( &num1, &num2);
   printf("Number1 = %d\n", num1);
   printf("Number2 = %d", num2);
   return 0;
}
void swap(int * n1, int * n2)
{
   // pointer n1 and n2 points to the address of num1 and num2 respectively
   int temp;
   temp = *n1;
   *n1 = *n2;
   *n2 = temp;
}
```

Output

Number1 = 10

Number2 = 5

Pointer Arithmetic

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

Incrementing a Pointer

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array –

```
#include <stdio.h>
const int MAX = 3;
int main () {
  int var[] = {10, 100, 200};
  int i, *ptr;
  /* let us have array address in pointer */
  ptr = var;
  for ( i = 0; i < MAX; i++) {
    printf("Address of var[%d] = %x\n", i, ptr );
    printf("Value of var[%d] = %d\n", i, *ptr );
    /* move to the next location */
    ptr++;
  }
  return 0;
}
```

Output

```
Address of var[0] = bf882b30
Value of var[0] = 10
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

Decrementing a Pointer

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below –

```
#include <stdio.h>
const int MAX = 3;
int main () {
  int var[] = {10, 100, 200};
  int i, *ptr;
  /* let us have array address in pointer */
  ptr = &var[MAX-1];
  for ( i = MAX; i > 0; i--) {
    printf("Address of var[%d] = %x\n", i-1, ptr );
    printf("Value of var[%d] = %d\n", i-1, *ptr );
    /* move to the previous location */
    ptr--;
  }
  return 0;
}
```

Output

```
Address of var[2] = bfedbcd8
Value of var[2] = 200
Address of var[1] = bfedbcd4
Value of var[1] = 100
Address of var[0] = bfedbcd0
Value of var[0] = 10
```

Dynamic Memory Allocation: Memory Allocation Functions

Dynamic memory allocation allows your program to obtain more memory space while running, or to release it if it's not required.

In simple terms, Dynamic memory allocation allows you to manually handle memory space for your program.

Although, C language inherently does not have any technique to allocate memory dynamically, there are 4 library functions under "stdlib.h" for dynamic memory allocation.

C malloc()

The name malloc stands for "memory allocation".

The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.

Syntax of malloc()

ptr = (cast-type*) malloc(byte-size)

Here, *ptr* is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

ptr = (int*) malloc(100 * sizeof(int));

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

C calloc()

The name calloc stands for "contiguous allocation".

The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Syntax of calloc()

ptr = (cast-type*)calloc(n, element-size);

This statement will allocate contiguous space in memory for an array of *n* elements. For example:

ptr = (float*) calloc(25, sizeof(float));

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

C free()

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on its own. You must explicitly use free() to release the space.

syntax of free()

free(ptr);

This statement frees the space allocated in the memory pointed by ptr.

Example #1: Using C malloc() and free()

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.

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

```
int main()
{
   int num, i, *ptr, sum = 0;
   printf("Enter number of elements: ");
   scanf("%d", &num);
   ptr = (int*) malloc(num * sizeof(int)); //memory allocated using malloc
   if(ptr == NULL)
   {
      printf("Error! memory not allocated.");
      exit(0);
   }
   printf("Enter elements of array: ");
   for(i = 0; i < num; ++i)
   {
      scanf("%d", ptr + i);
      sum += *(ptr + i);
   }
   printf("Sum = %d", sum);
   free(ptr);
   return 0;
}
```

C realloc()

If the previously allocated memory is insufficient or more than required, you can change the previously allocated memory size using realloc().

Syntax of realloc()

```
ptr = realloc(ptr, newsize);
```

Here, *ptr* is reallocated with size of newsize.

Example

```
#include  <stdio.h>
#include <stdlib.h>
int main()
{
   int *ptr, i , n1, n2;
```

```
    printf("Enter size of array: ");
    scanf("%d", &n1);
    ptr = (int*) malloc(n1 * sizeof(int));
    printf("Address of previously allocated memory: ");
    for(i = 0; i < n1; ++i)
        printf("%u\t",ptr + i);
    printf("\nEnter new size of array: ");
    scanf("%d", &n2);
    ptr = realloc(ptr, n2);
    for(i = 0; i < n2; ++i)
        printf("%u\t", ptr + i);
    return 0;
}
```

Programming Applications

Pointer is used for different purposes. Pointer is low level construct in programming which is used to perform high level task. Some of the pointer applications are listed below

A. Passing Parameter by Reference

First pointer application is to pass the variables to function using pass by reference scheme.

```
void interchange(int *num1,int *num2)
{
   int temp;
   temp = *num1;
   *num1 = *num2;
   *num2 = *num1;
}
```

Pointer can be used to simulate passing parameter by reference. Pointer is used to pass parameter to function. In this scheme we are able to modify value at direct memory location.

Re-commanded Article : Ways of function calling

B. Accessing Array element

```
int main()
{
```

```
int a[5] = {1,2,3,4,5};
int *ptr;
ptr = a;
for(i=0;i<5;i++) {
 printf("%d",*(ptr+i));
}
return(0);
}
```

We can access array using pointer. We can store base address of array in pointer.

ptr = a;

Now we can access each and individual location using pointer.

```
for(i=0;i<5;i++) {
    printf("%d",*(ptr+i));
}
```

C. Dynamic Memory Allocation :

Another pointer application is to allocate memory dynamically.

We can use pointer to allocate memory dynamically. Malloc and calloc function is used to allocate memory dynamically.

```
#include  <stdio.h>
#include <stdlib.h>
int main()
{
  char *str;
  str = (char *) malloc(15);
  strcpy(str, "mahesh");
  printf("String = %s, Address = %u\n", str, str);
  free(str);
  return(0);
}
```

consider above example where we have used malloc() function to allocate memory dynamically.

D. Reducing size of parameter

```
struct student {
  char name[10];
  int rollno;
};
```

Suppose we want to pass the above structure to the function then we can pass structure to the function using pointer in order to save memory.

Suppose we pass actual structure then we need to allocate (10 + 4 = 14 Bytes(*)) of memory. If we pass pointer then we will require 4 bytes(*) of memory.

E. Command-line Arguments.

main() function of a C program accepts arguments from command line or from other shell scripts by following commands. They are,

- argc
- argv[]

where,

argc – Number of arguments in the command line including program name

argv[] – This is carrying all the arguments

- In real time application, it will happen to pass arguments to the main program itself. These arguments are passed to the main () function while executing binary file from command line.
- For example, when we compile a program (test.c), we get executable file in the name "test".
- Now, we run the executable "test" along with 4 arguments in command line like below.

Example

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) // command line arguments
{
if(argc!=5)
{
  printf("Arguments passed through command line " \
       "not equal to 5");
```

```
  return 1;
}
  printf("\n Program name : %s \n", argv[0]);
  printf("1st arg : %s \n", argv[1]);
  printf("2nd arg : %s \n", argv[2]);
  printf("3rd arg : %s \n", argv[3]);
  printf("4th arg : %s \n", argv[4]);
  printf("5th arg : %s \n", argv[5]);
return 0;
}
```

Output:

```
Program name : test
1st arg : this
2nd arg : is
3rd arg : a
4th arg : program
5th arg : (null)
```

STRUCTURES

Structure is commonly referred to as a user-defined data type. C's structures allow you to store multiple variables of any type in one place (the structure). A structure can contain any of C's data types, including arrays and other structures. Each variable within a structure is called a member of the structure. They can hold any number of variables, and you can make arrays of structures. This flexibility makes structures ideally useful for creating databases in C.

DECLARATION OF STRUCTURES

To declare a structure you must start with the keyword **struct** followed by the structure name or structure tag and within the braces the list of the structure's member variables. Note that the structure declaration does not actually create any variables. **Syntax:-**

```
struct structure-tag {
datatype variable1;
datatype variable2;
datype variable 3;
...
```

```
};
```

Example:-

Consider the student database in which each student has a roll number, name and course and the marks obtained. Hence to group this data with a structure-tag as **student**, we can have the declaration of structure as:

```
struct student {
int roll_no;
char name[20];
char course[20];
int marks_obtained ;
};
struct student stud1, stud2 ;
```

The second method is as follows:

```
Struct student {
int roll_no;
char name[20];
char course[20];
int marks_obtained ;
} stud1, stud2 ;
```

ACCESSING THE MEMBERS OF A STRUCTURE

Individual structure members can be used like other variables of the same type.

Structure members are accessed using the structure member operator (.), also called the dot operator, between the structure name and the member name.

Syntax:-

```
structurevariable. member-name;
```

Example:-

```
struct coordinate{
        int x;
        int y;
}first,second;
```

Thus, to have the structure named first refer to a screen location that has coordinates x=50, y=100, you could write as,

```
first.x = 50;
```

```
first.y = 100;
```

To display the screen locations stored in the structure second, you could write,

```
printf ("%d,%d", second.x, second.y);
```

The individual members of the structure behave like ordinary date elements and can be accessed accordingly.

Example:-

/*Program to store and retrieve the values from the student structure*/

```
#include<stdio.h>
struct student {
        int roll_no;
        char name[20];
        char course[20];
        int marks_obtained ;
};
main()
{
student s1 ;
printf("Enter the student roll number:");
scanf ("%d",&s1.roll_no);
printf("\nEnter the student name: ");
scanf ("%s",s1.name);
printf("\nEnter the student course");
scanf ("%s",s1.course);
printf("Enter the student percentage\n");
scanf ("%d",&s1.marks_obtained);
printf ("\nData entry is complete");
printf ( "\nThe data entered is as follows:\n");
printf("\nThe student roll no is %d",s1.roll_no);
printf ("\nThe student name is %s",s1.name);
printf ("\nThe student course is %s",s1.course);
printf ("\nThe student percentage is %d",s1.marks_obtained);
}
```

OUTPUT

Enter the student roll number: 1234

Enter the student name: ARUN

Enter the student course: MCA

Enter the student percentage: 84

Data entry is complete

The data entered is as follows:

The student roll no is 1234

The student name is ARUN

The student course is MCA

The student percentage is 84

INITIALIZING STRUCTURES

Like other C variable types, structures can be initialized when they're declared. This procedure is similar to that for initializing arrays. The structure declaration is followed by an equal sign and a list of initialization values is separated by commas and enclosed in braces. For example, look at the following statements for initializing the values of the members of the mysale structure variable.

Example 9.2

```
struct sale {
char customer[20];
char item[20];
float amt;
    } mysale = { "XYZ Industries", "toolskit", 600.00 };
```

In a structure that contains structures as members, list the initialization values in order. They are placed in the structure members in the order in which the members are listed in the structure definition. Here's an example that expands on the previous one:

Example

```
struct customer {
char firm[20];
char contact[25];
}
struct sale {
struct customer buyer1;
char item [20];
float amt;
} mysale = { { "XYZ Industries", "Tyran Adams"}, "toolskit",600.00};
```

These statements perform the following initializations:

• the structure member mysale.buyer1.firm is initialized to the string “XYZ Industries”.

• the structure member mysale.buyer1.contact is initialized to the string “Tyran Adams”.

• the structure member mysale.item is initialized to the string "toolskit".

• the structure member mysale.amount is initialized to the amount 600.00.

For example let us consider the following program where the data members are initialized to some value.

Example

Write a program to access the values of the structure initialized with some initial values.

```
/* Program to illustrate to access the values of the structure initialized with some initial values*/
#include<stdio.h>
struct telephone{
int tele_no;
int cust_code;
char cust_name[20];
char cust_address[40];
int bill_amt;
};
main()
{
struct telephone tele = {2314345, 5463, "Ram", "New Delhi", 2435 };
printf("The values are initialized in this program.");
printf("\nThe telephone number is %d",tele.tele_no);
printf("\nThe customer code is %d",tele.cust_code);
printf("\nThe customer name is %s",tele.cust_name);
printf("\nThe customer address is %s",tele.cust_address);
printf("\nThe bill amount is %d",tele.bill_amt);
}
```

OUTPUT

```
The values are initialized in this program.
The telephone number is 2314345
The customer code is 5463
The customer name is Ram
The customer Address is New Delhi
The bill amount is 2435
```

STRUCTURES AS FUNCTION ARGUMENTS

C is a structured programming language and the basic concept in it is the modularity of the programs. This concept is supported by the functions in C language. This can be achieved in primarily two ways: Firstly, to pass them as simple parameter values by passing the structure name and secondly, through pointers.

Example

Write a program to demonstrate passing a structure to a function.

```
/*Program to demonstrate passing a structure to a function.*/
        #include <stdio.h>
        /*Declare and define a structure to hold the data.*/
        struct data{
                float amt;
                char fname [30];
                char lname [30];
            } person;
        main()
        {
            void print_person (struct data x); //function prototype
            printf("Enter the donor's first and last names separated by a space:");
            scanf ("%s %s", person.fname, person.lname);
            printf("\nEnter the amount donated in rupees:");
            scanf ("%f", &person.amt);
            print_person (person);
            return 0;
        }
        void print_person(struct data x)
        {
            printf ("\n %s %s gave donation of amount Rs.%.2f.\n", x.fname, x.lname,
            x.amt);
        }
```

OUTPUT

```
Enter the donor's first and last names separated by a space: RGUKT
Enter the amount donated in rupees: 1000.00
RGUKT gave donation of the amount Rs. 1000.00.
```

Example

Write a program to accept the data from the user and calculate the salary of the person using concept of functions.

```
/* Program to accept the data from the user and calculate the salary of the person*/
        #include<stdio.h>
        main()
        {
            struct sal {
                    char name[30];
                    int no_days_worked;
                    int daily_wage;
                };
            struct sal salary;
            struct sal get_dat(struct); /* function prototype*/
            float wages(struct); /*function prototype*/
            float amount_payable; /* variable declaration*/
            salary = get_data(salary);
            printf("The name of employee is %s",salary.name);
            printf("Number of days worked is %d",salary.no_daya_worked);
            printf("The daily wage of the employees is %d",salary.daily_wage);
            amount_payable = wages(salary);
            printf("The amount payable to %s is %f",salary.name,amount_payable);
            }
            struct sal get_data(struct sal income)
            {
                printf("Please enter the employee name:\n");
                scanf("%s",income.name);
                printf("Please enter the number of days worked:\n");
                scanf("%d",&income.no_days_worked);
                printf('Please enter the employee daily wages:\n");
                scanf("%d",&income.daily_wages);
                return(income);
            }
            float wages(struct)
```

```
{
        struct sal amt;
        int total_salary ;
        total_salary = amt.no_days_worked * amt.daily_wages;
        return(total_salary);
}
```

STRUCTURES AND ARRAYS

Structure is used to store the information of One particular object but if we need to store such 100 objects then Array of Structure is used.

Let's consider the example of students and their marks. In this case, to avoid declaring various data variables, we grouped together all the data concerning the student's marks as one unit and call it student. The problem that arises now is that the data related to students is not going to be of a single student only. We will be required to store data for a number of students. To solve this situation one way is to declare a structure and then create sufficient number of variables of that structure type. But it gets very burden to manage such a large number of data variables, so a better option is to declare an array.

So, revising the array for a few moments we would refresh the fact that an array is simply a collection of homogeneous data types. Hence, if we make a declaration as:

int temp[20];

It simply means that temp is an array of twenty elements where each element is of type integer, indicating homogenous data type.

```
struct student stud[20] ;
```

It means that stud is an array of twenty elements where each element is of the type struct student (which is a user-defined data type we had defined earlier). The various members of the stud array can be accessed in the similar manner as that of any other ordinary array.

Example:-

struct student stud[20], we can access the roll_no of this array as

```
stud[0].roll_no;
stud[1].roll_no;
stud[2].roll_no;
stud[3].roll_no; …
…
…
stud[19].roll_no;
```

Example:-

Write a program to read and display data for 20 students.

```
/*Program to read and print the data for 20 students*/
        #include <stdio.h>
        struct student {
                        int roll_no;
                        char name[20];
                        char course[20];
                        int marks_obtained ;
                    };
        main( )
        {
            struct student stud [20];
            int i;
            printf("Enter the student data one by one\n");
            for(i=0; i<=19; i++)
            {
            printf("Enter the roll number of %d student",i+1);
            scanf ("%d",&stud[i].roll_no);
            printf("Enter the name of %d student",i+1);
            scanf ("%s",stud[i].name);
            printf("Enter the course of %d student",i+1);
            scanf ("%d",stud[i].course);
            printf("Enter the marks obtained of %d student",i+1);
            scanf ("%d",&stud[i].marks_obtained);
        }
        printf("the data entered is as follows\n");
        for (i=0;i<=19;i++)
        {
            printf("The roll number of %d student is %d\n",i+1,stud[i].roll_no);
            printf ("The name of %d student is %s\n",i+1,stud[i].name);
            printf ("The course of %d student is %s\n",i+1,stud[i].course);
            printf ("The marks of %d student is %d\n",i+1,stud[i].marks_obtained);
        }
```

```
}
```

Example

/*Program to read and print data related to five students having marks of three subjects each using the concept of arrays */

```
#include<stdio.h>
struct student {
                int roll_no;
                char name [20];
                char course [20];
                int subject [3] ;
        };
main( )
{
    struct student stud[5];
    int i,j;
    printf(“Enter the data for all the students:\n”);
    for (i=0;i<=4;i++)
    {
    printf(“Enter the roll number of %d student”,i+1);
    scanf (“%d”,&stud[i].roll_no);
    printf(“Enter the name of %d student”,i+1);
    scanf (“%s”,stud[i].name);
    printf(“Enter the course of %d student”,i+1);
    scanf (“%s”,stud[i].course);
    for (j=0;j<=2;j++)
    {
        printf (“Enter the marks of the %d subject of the student
        %d:\n”,j+1,i+1);
        scanf (“%d”,&stud[i].subject[j]);
    }
}
printf(“The data you have entered is as follows:\n”);
for (i=0;i<=4;i++)
{
```

```
        printf("The %d th student's roll number is %d\n",i+1,stud[i].roll_no);
        printf ("The %d the student's name is %s\n",i+1,stud[i].name);
        printf("The %d the student's course is %s\n",i+1,stud[i].course);
        for (j=0;j<=2;j++)
{
        printf ("The %d the student's marks of %d I subject are %d\n",i+1, j+1,
        stud[i].subject[j]);
        }
}
printf ("End of the program\n");
}
```

UNIONS

C Union is also like structure, i.e. collection of different data types which are grouped together. Each element in a union is called member.

- Union and structure in C are same in concepts, except allocating memory for their members.
- Structure allocates storage space for all its members separately.
- Whereas, Union allocates one common storage space for all its members
- We can access only one member of union at a time. We can't access all member values at the same time in union. But, structure can access all member values at the same time. This is because, Union allocates one common storage space for all its members. Whereas Structure allocates storage space for all its members separately.
- Many union variables can be created in a program and memory will be allocated for each union variable separately.
- Below table will help you how to form a C union, declare a union, initializing and accessing the members of the union.

Syntax:-

```
union union-tag {
datatype variable1;
datatype variable2;
...
};
For example,
union temp{
```

```
int x;
char y;
float z;
};
```

In this case a float is the member which requires the largest space to store its value hence the space required for float (8 bytes) is allocated to the union. All members share the same space. Let us see how to access the members of the union.

Example

Write a program to illustrate the concept of union.

```
/* Declare a union template called tag */
union tag {
int nbr;
char character;
}
/* Use the union template */
union tag mixed_variable;
/* Declare a union and instance together */
union generic_type_tag {
        char c;
        int i;
        float f;
        double d;
    } generic;
```

INITIALIZING AN UNION

Let us see, how to initialize a Union with the help of the following example:

Example

```
union date_tag {
        char complete_date [9];
        struct part_date_tag {
                char month[2];
                char break_value1;
                char day[2];
                char break_value2;
                char year[2];
```

} parrt_date;

}date = {“01/01/05”};

ACCESSING THE MEMBERS OF AN UNION

Individual union members can be used in the same way as the structure members, by using the member operator or dot operator (.). However, there is an important difference in accessing the union members. Only one union member should be accessed at a time. Because a union stores its members on top of each other, it’s important to access only one member at a time. Trying to access the previously stored values will result in erroneous output.

Example:-

```
#include <stdio.h>
#include <string.h>
union student
{
        char name[20];
        char subject[20];
        float percentage;
};
int main()
{
   union student record1;
   union student record2;
   // assigning values to record1 union variable
     strcpy(record1.name, "Raju");
     strcpy(record1.subject, "Maths");
     record1.percentage = 86.50;
     printf("Union record1 values example\n");
     printf(" Name      : %s \n", record1.name);
     printf(" Subject   : %s \n", record1.subject);
     printf(" Percentage : %f \n\n", record1.percentage);
   // assigning values to record2 union variable
     printf("Union record2 values example\n");
     strcpy(record2.name, "Mani");
     printf(" Name      : %s \n", record2.name);
     strcpy(record2.subject, "Physics");
```

```
    printf(" Subject    : %s \n", record2.subject);
    record2.percentage = 99.50;
    printf(" Percentage : %f \n", record2.percentage);
    return 0;
}
```

FILE HANDLING IN C PROGRAMING

In C programming, file is a place on your physical disk where information is stored.

Uses of the files:

- When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.
- If you have to enter a large number of data, it will take a lot of time to enter them all. However, if you have a file containing all the data, you can easily access the contents of the file using few commands in C.
- You can easily move your data from one computer to another without any changes.

Types of Files

When dealing with files, there are two types of files you should know about:

1. Text files
2. Binary files

1. Text files

Text files are the normal .txt files that you can easily create using Notepad or any simple text editors. When you open those files, you'll see all the contents within the file as plain text. You can easily edit or delete the contents. They take minimum effort to maintain, are easily readable, and provide least security and takes bigger storage space.

2. Binary files

Binary files are mostly the .bin files in your computer. Instead of storing data in plain text, they store it in the binary form (0's and 1's). They can hold higher amount of data, are not readable easily and provides a better security than text files.

File Operations

In C, you can perform four major operations on the file, either text or binary:

1. Creating a new file
2. Opening an existing file
3. Closing a file
4. Reading from and writing information to a file

Working with files

When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and program.

FILE *fptr;

Opening a file - for creation and edit

Opening a file is performed using the library function in the **"stdio.h"** header file: fopen().

The syntax for opening a file in standard I/O is:

ptr = fopen("fileopen","mode")

For Example:

fopen("E:\\cprogram\\newprogram.txt","w");

fopen("E:\\cprogram\\oldprogram.bin","rb");

- Let's suppose the file newprogram.txt doesn't exist in the location E:\cprogram. The first function creates a new file named newprogram.txt and opens it for writing as per the mode 'w'. The writing mode allows you to create and edit (overwrite) the contents of the file.
- Now let's suppose the second binary file oldprogram.bin exists in the location E:\cprogram. The second function opens the existing file for reading in binary mode 'rb'. The reading mode only allows you to read the file, you cannot write into the file.

Opening Modes in Standard I/O

File Mode	Meaning of Mode	During Inexistence of file
r	Open for reading.	If the file does not exist, fopen() returns NULL.
rb	Open for reading in binary mode.	If the file does not exist, fopen() returns NULL.
w	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb	Open for writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a	Open for append. i.e, Data is added to end of file.	If the file does not exists, it will be created.
ab	Open for append in binary mode. i.e, Data is added to end of file.	If the file does not exists, it will be created.
r+	Open for both reading and writing.	If the file does not exist, fopen() returns NULL.

Opening Modes in Standard I/O

File Mode	Meaning of Mode	During Inexistence of file
rb+	Open for both reading and writing in binary mode.	If the file does not exist, fopen() returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb+	Open for both reading and writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exists, it will be created.
ab+	Open for both reading and appending in binary mode.	If the file does not exists, it will be created.

Closing a File

The file (both text and binary) should be closed after reading/writing. Closing a file is performed using library function fclose().

```
fclose(fptr);    //fptr is the file pointer associated with file to be closed.
```

Reading and writing to a text file

For reading and writing to a text file, we use the functions fprintf() and fscanf(). They are just the file versions of printf() and scanf(). The only difference is that, fprint and fscanf expects a pointer to the structure FILE.

Example 1: Write to a text file using fprintf()

```
#include <stdio.h>
int main()
{
  int num;
  FILE *fptr;
  fptr = fopen("C:\\program.txt","w");
  if(fptr == NULL)
  {
    printf("Error!");
    exit(1);
  }
```

```
  printf("Enter num: ");
  scanf("%d",&num);
  fprintf(fptr,"%d",num);
  fclose(fptr);
  return 0;
}
```

This program takes a number from user and stores in the file program.txt. After you compile and run this program, you can see a text file program.txt created in C drive of your computer. When you open the file, you can see the integer you entered.

Example 2: Read from a text file using fscanf()

```
#include <stdio.h>
int main()
{
  int num;
  FILE *fptr;
  if ((fptr = fopen("C:\\program.txt","r")) == NULL){
     printf("Error! opening file");
     // Program exits if the file pointer returns NULL.
     exit(1);
  }
  fscanf(fptr,"%d",&num);
  printf("Value of n=%d",num);
  fclose(fptr);
  return 0;
}
```

This program reads the integer present in the program.txt file and prints it onto the screen.
If you succesfully created the file from **Example 1**, running this program will get you the integer you entered. Other functions like fgetchar(), fputc() etc. can be used in similar way.

Reading and writing to a binary file

Functions fread() and fwrite() are used for reading from and writing to a file on the disk respectively in case of binary files.

Writing to a binary file

To write into a binary file, you need to use the function fwrite(). The functions takes four arguments: Address of data to be written in disk, Size of data to be written in disk, number of such type of data and pointer to the file where you want to write.

```
fwrite(address_data, size_data, numbers_data, pointer_to_file);
```

Example 3: Writing to a binary file using fwrite()

```
#include <stdio.h>
struct threeNum
{
  int n1, n2, n3;
};
int main()
{
  int n;
  struct threeNum num;
  FILE *fptr;
  if ((fptr = fopen("C:\\program.bin","wb")) == NULL){
     printf("Error! opening file");
     // Program exits if the file pointer returns NULL.
     exit(1);
  }
  for(n = 1; n < 5; ++n)
  {
    num.n1 = n;
    num.n2 = 5n;
    num.n3 = 5n + 1;
    fwrite(&num, sizeof(struct threeNum), 1, *fptr);
  }
  fclose(fptr);
  return 0;
}
```

In this program, you create a new file program.bin in the C drive. We declare a structure threeNum with three numbers - *n1, n2 and n3*, and define it in the main function as num.

Now, inside the for loop, we store the value into the file using fwrite. The first parameter takes the address of *num* and the second parameter takes the size of the structure threeNum.
Since, we're only inserting one instance of *num*, the third parameter is 1. And, the last parameter *fptr points to the file we're storing the data. Finally, we close the file.

Reading from a binary file

Function fread() also take 4 arguments similar to fwrite() function as above.

```
fread(address_data, size_data, numbers_data, pointer_to_file);
```

Example 4: Reading from a binary file using fread()

```
#include <stdio.h>
struct threeNum
{
  int n1, n2, n3;
};
int main()
{
  int n;
  struct threeNum num;
  FILE *fptr;
  if ((fptr = fopen("C:\\program.bin","rb")) == NULL){
     printf("Error! opening file");
     // Program exits if the file pointer returns NULL.
     exit(1);
  }
  for(n = 1; n < 5; ++n)
  {
    fread(&num, sizeof(struct threeNum), 1, *fptr);
    printf("n1: %d\tn2: %d\tn3: %d", num.n1, num.n2, num.n3);
  }
  fclose(fptr);
  return 0;
}
```

In this program, you read the same file program.bin and loop through the records one by one.

In simple terms, you read one threeNum record of threeNum size from the file pointed by **fptr* into the structure *num*.

Example 4:- Write a C program to Write a Sentence to a File

This program stores a sentence entered by user in a file.

```
#include <stdio.h>
#include <stdlib.h> /* For exit() function */
int main()
{
  char sentence[1000];
  FILE *fptr;
  fptr = fopen("program.txt", "w");
  if(fptr == NULL)
  {
    printf("Error!");
    exit(1);
  }
  printf("Enter a sentence:\n");
  gets(sentence);
  fprintf(fptr,"%s", sentence);
  fclose(fptr);
  return 0;
}
```

Output

Enter sentence:

I am awesome and so are files.

After termination of this program, you can see a text file program.txt created in the same location where this program is located.

If you open and see the content, you can see the sentence: I am awesome and so are files.

In this program, a file is opened using opening mode "w". In this mode, if the file exists, its contents are overwritten and if the file does not exist, it will be created. Then, user is asked to enter a sentence. This sentence will be stored in file program.txt using fprintf() function.

Example 5: Write a C program to read text from a file

```
#include <stdio.h>
#include <stdlib.h> // For exit() function
```

```
int main()
{
   char c[1000];
   FILE *fptr;
   if ((fptr = fopen("program.txt", "r")) == NULL)
   {
      printf("Error! opening file");
      // Program exits if file pointer returns NULL.
      exit(1);
   }
   // reads text until newline
   fscanf(fptr,"%[^\n]", c);
   printf("Data from the file:\n%s", c);
   fclose(fptr);
   return 0;
}
```

If the file program.txt is not found, this program prints error message. If the file is found, the program saves the content of the file to a string *c* until '\n' newline is encountered.

Suppose, the program.txt file contains following text.

C programming is awesome.

I love C programming.

How are you doing?

The output of the program will be:

Data from the file: C programming is awesome.

Example 6: Write a C program to read name and marks of n number of students from user and store them in a file.

```
#include <stdio.h>
int main()
{
  char name[50];
  int marks, i, num;
  printf("Enter number of students: ");
  scanf("%d", &num);
```

```
  FILE *fptr;
  fptr = (fopen("C:\\student.txt", "w"));
  if(fptr == NULL)
  {
     printf("Error!");
     exit(1);
  }
  for(i = 0; i < num; ++i)
  {
    printf("For student%d\nEnter name: ", i+1);
    scanf("%s", name);
    printf("Enter marks: ");
    scanf("%d", &marks);
    fprintf(fptr,"\nName: %s \nMarks=%d \n", name, marks);
  }
  fclose(fptr);
  return 0;
}
```

Example 7:- Write a C program to read name and marks of n number of students from user and store them in a file. If the file previously exits, add the information of n students.

```
#include <stdio.h>
int main()
{
  char name[50];
  int marks, i, num;
  printf("Enter number of students: ");
  scanf("%d", &num);
  FILE *fptr;
  fptr = (fopen("C:\\student.txt", "a"));
  if(fptr == NULL)
  {
     printf("Error!");
     exit(1);
  }
```

```
  for(i = 0; i < num; ++i)
  {
    printf("For student%d\nEnter name: ", i+1);
    scanf("%s", name);

    printf("Enter marks: ");
    scanf("%d", &marks);
    fprintf(fptr,"\nName: %s \nMarks=%d \n", name, marks);
  }
  fclose(fptr);
  return 0;
}
```

Example 8: Write a C program to write all the members of an array of strcures to a file using fwrite(). Read the array from the file and display on the screen.

```
#include <stdio.h>
struct student
{
  char name[50];
  int height;
};
int main(){
   struct student stud1[5], stud2[5];
   FILE *fptr;
   int i;
   fptr = fopen("file.txt","wb");
   for(i = 0; i < 5; ++i)
   {
      fflush(stdin);
      printf("Enter name: ");
      gets(stud1[i].name);
      printf("Enter height: ");
      scanf("%d", &stud1[i].height);
```

```
    }
    fwrite(stud1, sizeof(stud1), 1, fptr);
    fclose(fptr);
    fptr = fopen("file.txt", "rb");
    fread(stud2, sizeof(stud2), 1, fptr);
    for(i = 0; i < 5; ++i)
    {
       printf("Name: %s\nHeight: %d", stud2[i].name, stud2[i].height);
    }
    fclose(fptr);
}
```