C PROGRAMMING

Linkedin@KeerthanaGaliveeti

1. INTRODUCTION

The C Language is developed by Dennis Ritchie for creating system applications that directly interact with the hardware devices such as drivers, kernels, etc.
C programming is considered as the base for other programming languages, that is why it is known as the "**mother language**".

C language is considered as the mother language of all the modern programming languages because most of the compilers, **JVMs, Kernels**, etc. are written in **C language**, and most of the programming languages follow C syntax, for example, C++, Java, C#, etc.

It provides the core concepts like the **array, strings, functions, file handling**, etc. that are being used in many languages like C++, Java, C#, etc.
Online Tool : https://www.programiz.com/c-programming/online-compiler/

C as Procedural Language :

A procedure is known as a function, method, routine, subroutine, etc. A procedural language **specifies a series of steps for the program to solve the problem**.A procedural language breaks the program into functions, data structures, etc.C is a procedural language. In C, variables and function prototypes must be declared before being used.

1.1 SIMPLE PROGRAM

```
#include <stdio.h>
int main()  / void main()
{
printf("Hello C Programming\n");  -string
return 0;
}
```

Description :

#include<stdio.h> - Header File

int main () - Main function

printf() - printing statement to display the output

1.2 HISTORY OF C

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

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

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

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

2. FEATURES OF C

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

1. Simple
2. Machine Independent or Portable
3. Mid-level programming language
4. structured programming language
5. Rich Library
6. Memory Management
7. Fast Speed
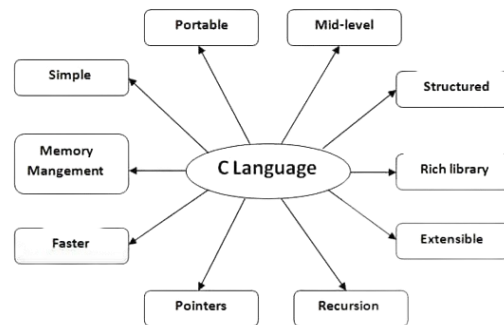
1) Simple

C is a simple language in the sense that it provides a **structured approach** (to break the problem into parts), **the rich set of library functions**, **data types**, etc. int num1=10; integer -int , floating-float str jayendra="50"; - string

2) Machine Independent or Portable

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

3) Mid-level programming language

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

4) Structured programming language

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

5) Rich Library

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

6) Memory Management

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

7) Speed

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

3. OVERVIEW - STRUCTURE OF PROGRAM IN C

Syntax :

    #Libraries(header files)

    Type Function name()

    {

    /*variable declaration*/

    #statements;

    #statements;

    }

Example :

    #include<stdio.h>

    int main()

    {

    int a=5; —variable initialization

    printf(a);

    return 0; }

Components :

1. Header files
2. Main function
3. Variable declaration
4. Statements and expressions
5. Comment
6. Functions
7. Return statement
8. Standard Input/Output

1. Header Files :

The #include directives at the beginning of the program are used to include header files. Header files provide function **prototypes** and **definitions** that allow the C compiler to understand the functions used in the program.

2. Main Function:

Every C program starts with the main function. It is the program's entry point, and execution starts from here. The main function has a return type of int, indicating that it should return an integer value to the operating system upon completion.

3. Variable Declarations:

Before using any variables, you should declare them with their *data types*. This section is typically placed after the main function's curly opening brace.

4. Statements and Expressions:

This section contains the actual instructions and logic of the program. C programs are composed of statements that perform actions and expressions that compute values.

5. Comments:

Comments are used to provide human-readable explanations within the code. They are not executed and do not affect the program's functionality. In C, comments are denoted by // for single-line comments and /* */ for multi-line comments.

6. Functions:

C programs can include *user-defined* functions and *blocks* of code that perform specific tasks. Functions help modularize the code and make it more organized and manageable.

7. Return Statement:

Use the return statement to terminate a function and return a value to the caller function. A return statement with a value of 0 typically indicates a successful execution in the main function, whereas a non-zero value indicates an error or unexpected termination.

8. Standard Input/Output:

C has library functions for reading user input (scanf) and printing output to the console (printf). These functions are found in C programs and are part of the standard I/O library (stdio.h header file). It is essential to include these fundamental features correctly while writing a simple C program to ensure optimal functionality and readability.

4. COMPILATION PROCESS

The compilation is a process of converting the source code into object code. It is done with the help of the compiler. The compiler checks the source code for the syntactical or structural errors, and if the source code is error-free, then it generates the object code.

The c compilation process converts the source code taken as input into the object code or machine code. The compilation process can be divided into four steps, i.e., Pre-processing, Compiling, Assembling, and Linking.
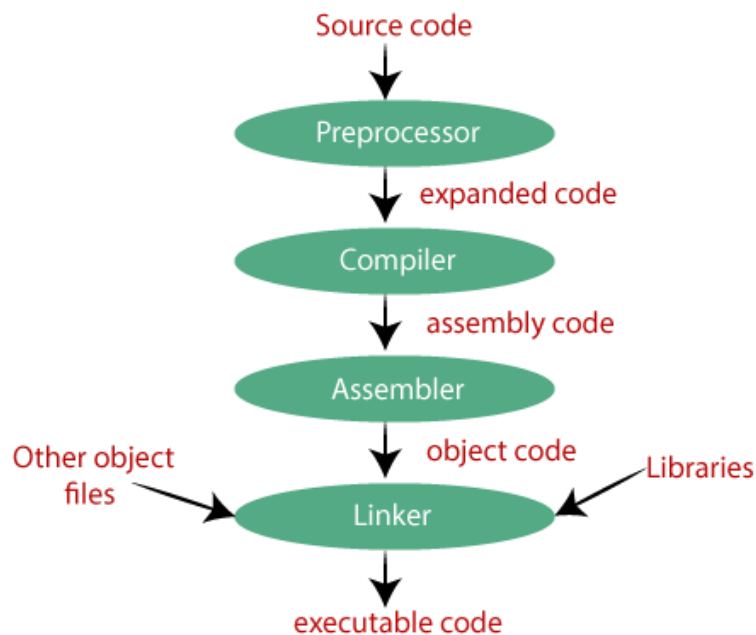
Various components involved in the compilation process as follows :

1. **Preprocessor**

2. **Compiler**

3. **Assembler**

4. **Linker**

Preprocessor (eg: add.c)   (eg: img.jpg)

The source code is the code which is written in a text editor and the source code file is given an extension ".c". This source code is first passed to the preprocessor, and then the preprocessor expands this code. After expanding the code, the expanded code is passed to the compiler.

Compiler

The code which is expanded by the preprocessor is passed to the compiler. The compiler converts this code into assembly code. Or we can say that the C compiler converts the pre-processed code into assembly code. $0's \& 1's$

Assembler add.c —>add.obj

The assembly code is converted into object code by using an assembler. The name of the object file generated by the assembler is the same as the source file. The extension of the object file in DOS is '.obj,'. If the name of the source file is **'hello.c',** then the name of the object file would be 'hello.obj'.

Linker

The main working of the linker is to combine the object code of library files with the object code of our program. Sometimes the situation arises when our program refers to the functions defined in other files; then linker plays a very important role in this. It links the object code of these files to our program. **Add.exe ->output**

Illustration

Program 1: (This program is saved as **"hello.c"**)

#include<stdio.h>

int main()

{

    printf("Hi I am a computer student");

}

Step 1: The input file, i.e., **hello.c,** is passed to the preprocessor, and the preprocessor converts the source code into expanded source code. The extension of the expanded source code would be **hello.i.**

Step 2: The expanded source code is passed to the compiler, and the compiler converts this expanded source code into assembly code. The extension of the assembly code would be **hello.s.**

Step 3: This assembly code is then sent to the assembler, which converts the assembly code into object code.

Step 4 : After the creation of an object code, the linker creates the executable file. The loader will then load the executable file for the execution.
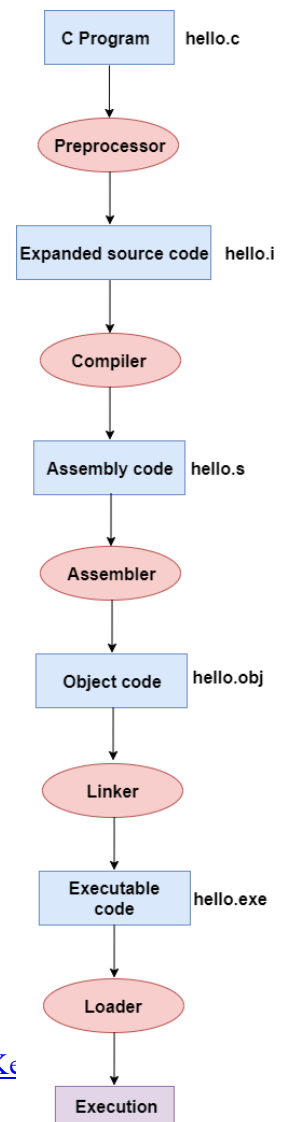
6. INPUT & OUTPUT OPERATIONS

There are two operations involved to get the input and display the output.

- printf()
- scanf()

printf() function :

This function is used to display the output produced.

Syntax :

printf("format string",argument_list);

——->**printf("%d",a);**

Format string - used to indicate which datatype, the variable belongs too. Eg: %s(string), %d(integer).,etc %c-char, %s-string, %f-float

scanf() function :

This function is used to get the input from the user. It reads the input from the console.

Syntax :

scanf("format string",argument_list);

----> **scanf("%d", &a);**

Example programs

Program 2:

```
#include<stdio.h>

int main()

{

int num1,num2;

printf("Enter first number:");

scanf("%d", &num1);

printf("Enter second number:");

scanf("%d", &num2);

printf(" The addition of two numbers:",num1+num2);

}
```

```
#include<stdio.h>

 int main()

 {

 int num1=5;

 print(num1);

 }
```

Output :

Enter first number:5

Enter second number:10

The addition of two numbers:15

7. VARIABLES IN C

A variable is the name of the memory location. It is used to store information.

Each variable has a ***unique identifier***, its ***name***, and a ***data type*** describing the type of data it may hold.

Rules for framing a var   iables :

- ***Variable names*** include ***letters (uppercase and lowercase), digits***, and ***underscores***. They must start with a ***letter (uppercase or lowercase)*** or an ***underscore***. (_)
- C is a ***case-sensitive*** programming language. It means that ***uppercase*** and ***lowercase letters*** are considered distinct. Eg : m1, M1 are considered as two different variables.
- ***Variable names*** cannot be the same as ***C keywords (reserved words)***, as they have special meanings in the language.
- ***Variable names*** cannot contain ***spaces*** or special characters (such as ***!, @, #, $, %, ^, &, *, (,), -, +, =, [,], {, }, |, \, /, <, >,., ?;, ', or ")***.

Variable Declaration :

The variables can be declared in the C program and the values can be assigned later.

Syntax :

<data_type> variable_name;   eg : int num1;

Here data_type mentions about the type of variable declared whether it is a integer/string/floating number/character.

Variable Initialization :

The variables can be assigned values by using assignment operator (=).

Syntax :

<data_type> variable_name = value; int num1=5;

Example programs :

Program 3 :

To find the square number :

#include<stdio.h>
int main ()

```
{
    int a=2;
    print("The square number of the number is :", a*a);
}
```

Output :

Enter number : 2

The square root of the number is : 4

Keywords:

1. Int, printf,scanf,main(),include,return,void
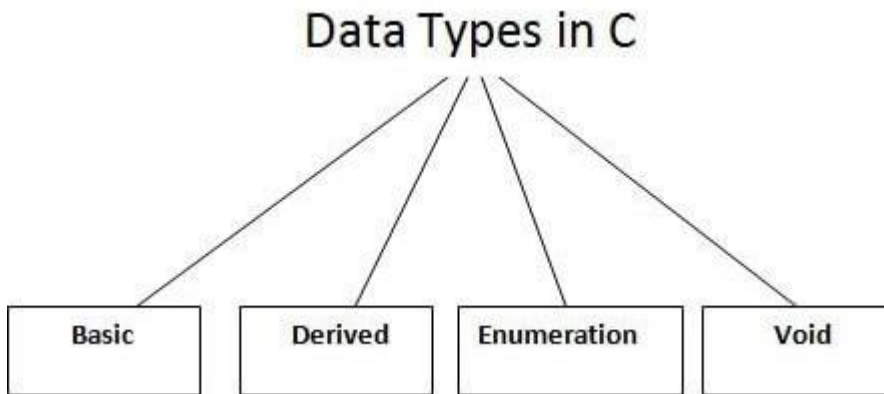
2. Case-sensitive

   int a; int A;

   Constants :

   Pi =22/7 or 3.14

8. DATATYPES IN C

A data type specifies the type of data that a variable can store such as integer, floating, character, etc.

Data Types in C

Basic	Derived	Enumeration	Void

Basic Data Type	int, char, float, double
Derived Data Type	array, pointer, structure, union
Enumeration Data Type	enum
Void Data Type	void

Basic Data Types

The basic data types are integer-based and floating-point based. C language supports both signed and unsigned literals.

The memory size of the basic data types may change according to 32 or 64-bit operating system.

Int: - %d

Integers are entire numbers without any fractional or decimal parts, and the *int data type* is used to represent them.

It is frequently applied to variables that include *values*, such as *counts, indices*, or other numerical numbers. The *int data type* may represent both *positive* and *negative numbers* because it is signed by default.

Char: -%c

Individual characters are represented by the *char data type*. Typically used to hold *ASCII* or *UTF-8 encoding scheme characters*, such as *letters, numbers, symbols*, or *commas*. There are *256 characters* that can be represented by a single char, which takes up one byte of memory. Characters such as *'A', 'b', '5',* or *'$'* are enclosed in single quotes.

char a='1';   char a[]="apple"; -%s

Float: -%f

To represent integers, use the floating data type. Floating numbers can be used to represent fractional units or numbers with decimal places.

The float type is usually used for variables that require very good precision but may not be very precise. It can store values with an accuracy of about 6 decimal places and a range of about 3.4 x 1038 in 4 bytes of memory. 3.140000

Double: -%lf

Use two data types to represent *two floating integers*. When additional precision is needed, such as in scientific calculations or financial applications, it provides greater accuracy compared to float. 3.1400000000000

Double type, which uses *8 bytes* of memory and has an accuracy of about *15 decimal places, yields larger values*. C treats floating point numbers as doubles by default if no explicit type is supplied.

1. **int** age = 25; / temp=-90
2. **char** grade = 'A';
3. **float** temperature = 98.6;
4. **double** pi = 3.14159265359;

Derived Data Type

Beyond the fundamental data types, C also supports *derived data types,* including *arrays, pointers, structures,* and *unions*.

<div align="center">

0 1 2 3 4 5   print("%d", a[2]);

</div>

Array: int a[6]={1,1,2,3,4,5}

An *array, a derived data type*, lets you store a sequence of *fixed-size elements* of the same type. It provides a mechanism for joining multiple targets of the same data under the same name.

The index is used to access the elements of the array, with a ***0 index*** for the first entry. The size of the array is fixed at declaration time and cannot be changed during program execution.

Pointer:

A ***pointer*** is a derived data type that keeps track of another data type's memory address. When a ***pointer*** is declared, the ***data type*** it refers to is ***stated first***, and then the ***variable name*** is preceded by ***an asterisk (*).*** ;

Structure:

A structure is a derived data type that enables the creation of composite data types by allowing the grouping of many data types under a single name. It gives you the ability to create your own unique data structures by fusing together variables of various sorts.

1. A structure's members or fields are used to refer to each variable within it.
2. Any data type, including different structures, can be a member of a structure.
3. A structure's members can be accessed by using the dot (.) operator.

Union:

A derived data type called a union enables you to store various data types in the same memory address. In contrast to structures, where each member has a separate memory space, members of a union all share a single memory space. A value can only be held by one member of a union at any given moment. When you need to represent many data types interchangeably, unions come in handy. Like structures, you can access the members of a union by using the dot (.) operator.

Enumeration Data Type

A set of named constants or *enumerators* that represent a collection of connected values can be defined in C using the *enumeration data type (enum). Enumerations* give you the means to give names that make sense to a group of integral values, which makes your code easier to read and maintain.

Void Data Type

The *void data type* in the C language is used to denote the lack of a particular type. *Function return types, function parameters*, and *pointers* are three situations where it is frequently utilized.

SAMPLE PROGRAMS :

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

 int main()

 {

   int num1 = 5;

   int num2 = 10;

   int sum;

   sum = num1 + num2;

   printf("Sum of %d and %d is %d\n", num1, num2, sum);

   return 0;

}
```

2.
```c
#include <stdio.h>

int main() {

float num1 = 3.5;

float num2 = 2.5;

float product;

product = num1 * num2;

printf("Product of %.2f and %.2f is %.2f\n", num1, num2, product);

return 0;

}
```

3.
```c
#include <stdio.h>

int main()

{

char letter = 'A';

printf("The character is: %c\n", letter);

return 0;

}
```

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

int main()

{

    char name[] = "John Doe";

    printf("Name: %s\n", name);

    return 0;

}
```

**ARRAY :**

```c
#include <stdio.h>

int main() {

    // Declare an array of integers
```

OUTPUT :

1

2

int numbers[] = {1, 2, 3, 4, 5};   char name[3]={'a', 'b','c'};

                    0 1 2 3 4

// Print the array elements without using a loop

printf("Array elements: ");

printf("%d %d %d %d %d\n", numbers[0], numbers[1], numbers[2], numbers[3], numbers[4]);

return 0;

}

POINTER :

#include <stdio.h>

int main() {

    // Declare an integer variable and a pointer to it

    int number = 10;

    int *a = &number;

    // Print the value of the integer variable and its address using the pointer

    printf("Value of number: %d\n", *a);

    printf("Address of number: %p\n", a);

```c
    return 0;

}
```

ENUM :

```c
#include <stdio.h>
enum Week {Monday,  Tuesday,  Wednesday,  Thursday };
int main()
 {
    enum Week today;
     today =Wednesday;
     printf("Today is %d\n", today);
   return 0;
}
```

STRUCTURE:

```c
#include <stdio.h>
struct Point
{
   int x;
   int y;
};
   int main() {
   struct Point p1 = {3, 5};
   printf("Coordinates of the point: (%d, %d)\n", p1.x, p1.y);
   return 0;
```

}

9. C OPERATORS

An operator is simply a symbol that is used to perform operations. There can be many types of operations like arithmetic, logical, bitwise, etc.

- Arithmetic Operators   → +,-,*,/,%

- Relational Operators

- Shift Operators

- Logical Operators

- Bitwise Operators

- Ternary or Conditional Operators

- Assignment Operator

- Miscellaneous operators

Arithmetic Operators:

Arithmetic operators carry out *fundamental mathematical* operations. The arithmetic operators in C are as follows:

Addition Operator (+): The addition operator *adds* two operands together.

Syntax:

result = operand1 + operand2;

1. **int** a = 5;
2. **int** b = 3;
3. **int** result = a + b;   4/2=2   4%2=0

Relational Operators:

Relational operators assess the relationship between values by comparing them. They return either *true (1)* or *false (0)*. The relational operators in C are as follows:

1. **Equality Operator (==):** If two operands are equal, the *equality operator* verifies this.

Syntax:

```
result = operand1 == operand2;
```

Example:

1. int a = 5;
2. int b = 5;
3. int result = a == b;

Output:

result=1 (true)

2. Inequality Operator (!=):

The *inequality operator* determines whether two operands are *equal* or *not.*

Syntax:

It has the following syntax:

1. result = operand1 != operand2;

Example:

1. int a = 5;
2. int b = 5;
3. int result = a != b;

Output:

result=0 (False)

3. Greater than Operator (>):

The *greater than operator* determines if the first operand exceeds the second operand.

Syntax:

It has the following syntax:

1. result = operand1 > operand2;

Example:

1. int a = 7;
2. int b = 4;
3. int result = a > b;

Output:

result=1 (true)

4. Less than Operator (<):

The *less-than operator* determines if the first operand less is than the second operand.

Syntax:

It has the following syntax:

1. result = operand1 < operand2;

Example:

1. int a = 2;
2. int b = 6;
3. int result = a < b;

Output:

result=1 (true)

Greater than or Equal to Operator (>=):

The *greater than or equal to operator* determines if the first operand is more than or equal to the second operand.

Syntax:

It has the following syntax:

1. result = operand1 >= operand2;

Example:

1. int a = 15;
2. int b = 15;
3. int result = a >= b; 14>=15

Output:

result=1(t)

Less than or Equal To Operator (<=):

The *less than or equal to operator* determines if the first operand must be less than or equal to the second operand.

Syntax:

It has the following syntax:

1. result = operand1 <= operand2;

Example:

1. int a = 3;
2. int b = 6;
3. int result = a <= b; 3<=6

Output:

Result =1(true)

Shift Operators:

A binary number's bits can be moved to the *left* or *right* using *shift operators*. The C shift workers are listed below:

Left Shift Operator (<<): The *left shift operator* moves the bits of the first operand to the left by the number of places indicated by the second argument.

Syntax:

It has the following syntax:

1. result = operand1 << operand2;

Example:

1. unsigned **int** a = 5; // 0000 0101 in binary   00010100 - 20
2. **int** result = a << 2;   // 01010000

Output:

result = 20 // 0001 0100 in binary

Right Shift Operator (>>): The *right shift operator* shifts the bits of the first operand to the *right* by the number of positions specified by the second operand.

Syntax:

It has the following syntax:

1. result = operand1 >> operand2;

Example:

1. unsigned **int** a = 20; // 0001 0100 in binary

2. **int** result = a >> 2;   // 00000101 - 5

Output:

result = 5 // 0000 0101 in binary

Logical Operators:

Logical operators perform logical operations on *boolean values* and return either *true (1)* or *false (0).* Here are the logical operators in C:

Logical AND Operator (&&): The *logical AND operator* returns *true* if both operands are *true*.

Syntax:

It has the following syntax:

1. result = operand1 && operand2;

Example:

1. **int** a = 5;
2. **int** b = 3;
3. **int** result = (a > 3) && (b < 5);

Output:

result = 1 (true)

Logical OR Operator (||): The *logical OR operator* returns *true* if at least one of the operands is true.

Syntax:

It has the following syntax:

1. result = operand1 || operand2;

Example:

1. **int** a = 5;
2. **int** b = 3;
3. **int** result = (a > 3) || (b > 5);

Output:

result = 1 (true)

Logical NOT Operator (!): The *logical NOT operator* negates the value of the operand.

Syntax:

It has the following syntax:

1. result = !operand;

Example:

1. **int** a = 5;
2. **int** result = !(a < 3);

Output:

result = 1 (true)

Bitwise Operators:

Bitwise operators perform operations on individual *bits* of the operands. Here are the bitwise operators in C:

Bitwise AND Operator (&): The **bitwise AND operator** performs a *bitwise AND operation* on the corresponding bits of the operands.

Syntax:

It has the following syntax:

1. result = operand1 & operand2;

Example:

1. unsigned **int** a = 5; // 0000 0101 in binary
2. unsigned **int** b = 3; // 0000 0011 in binary
3. **int** result = a & b;

Output:

result = 1 // 0000 0001 in binary

Bitwise OR Operator (|): The *bitwise OR operator* performs a bitwise OR operation on the corresponding bits of the operands.

Syntax:

It has the following syntax:

1. result = operand1 | operand2;

Example:

1. unsigned **int** a = 5; // 0000 0101 in binary
2. unsigned **int** b = 3; // 0000 0011 in binary
3. **int** result = a | b;   00000111

Output:

result = 7 // 0000 0111 in binary

Bitwise XOR Operator (^): The *bitwise XOR operator* performs a bitwise exclusive OR operation on the corresponding bits of the operands.

Syntax:

It has the following syntax:

1. result = operand1 ^ operand2;   when both operands have 1 the result -0

Example:

1. unsigned **int** a = 5; // 0000 0101 in binary   0010
2. unsigned **int** b = 3; // 0000 0011 in binary   1110
3. **int** result = a ^ b;   1100

Output:

result = 6 // 0000 0110 in binary

Bitwise NOT Operator (~): The *bitwise NOT operator* flips each bit of the operand.

Syntax:

It has the following syntax:

1. result = ~operand;

Example:

1. unsigned **int** a = 5; // 0000 0101 in binary
2. **int** result = ~a; 11111010-   250

Output:

result = 250 // 1111 1001 in binary (assuming 8-bit representation)

Ternary or Conditional Operator: The *ternary or conditional operator* allows you to *assign* a value based on a condition.

Syntax:

It has the following syntax:

1. result = condition ? value1 : value2;

Example:

1. **int** a = 5;
2. **int** b = 3;   5<3   5   3
3. **int** result = (a <b) ? a : b;

Output:

result = 3

Assignment Operator:

Assignment operators are used to assign values to variables. Here is some of the assignment operator in C:

Simple Assignment Operator (=): The *simple assignment operator* assigns the value from the *right* side operands to the *left side operands*.

Syntax:

It has the following syntax:

1. variable = value;

Example:

1. **int** a; —variable declaration
2. a = 5; –variable initialization

Output:

No output. The value 5 is assigned to variable 'a'.

Miscellaneous Operator:

The *sizeof operator* and the *comma operator* fall under the *miscellaneous operator category*.

sizeof Operator: The *sizeof operator* returns the size, in *bytes*, of a *variable* or a *data type*.

Syntax: **int- 4 bytes, float-4 bytes,char- 1 byte, double -8bytes;**

It has the following syntax:

1. result = **sizeof**(variable / data type);

Example:

1. **int** a;
2. **int** size = **sizeof**(a);

Output:

size = 4 // Assuming int occupies 4 bytes

Comma Operator (,): The *comma operator* evaluates multiple expressions and returns the value of the *last expression*.

Syntax:

It has the following syntax:

1. makefileCopy code
2. result = (expression1, expression2,..., expressionN);

Example:

1. **int** a = 5, b = 3; a=a+2 , b=b*2,a+b
2. **int** result = (a += 2, b *= 2, a + b);

Increment / Decrement operator :

Increment adding 1

Decrement subtracting 1

a+=1; —> a=a+1;   a++ => a=a+1

a-=5; —> a=a-5 a=10   o/p : 5 b*=10 → b=b*10

10. CONTROL STATEMENTS

The if-else statement in C is used to perform the operations based on some specific condition. The operations specified in the if block are executed if and only if the given condition is true.

There are the following variants of the if statement in C language.

- ○ If statement
- ○ If-else statement

- ○ If else-if ladder
- ○ Nested if

If Statement

The if statement is used to check some given condition and perform some operations depending upon the correctness of that condition. It is mostly used in the scenario where we need to perform the different operations for the different conditions. The syntax of the if statement is given below.

1. if(expression){

2. //code to be executed

3. }

Flowchart of if statement in C

```
if(expression) {

    //code to be executed

}
else {

    //code to be executed

}
```

```
if(num1%2==0){

    printf("even");

}
else {

    printf("odd");

}
```

Let's see a simple example of the C language if statement.

```c
#include<stdio.h>

int main(){

int number=0;

printf("Enter a number:");

scanf("%d",&number);

if(number%2==0){
```

printf("%d is even number",number);

}

return 0;

}

Output

Enter a number:4

4 is even number

enter a number:5

If else-if ladder Statement

The if-else-if ladder statement is an extension to the if-else statement. It is used in the scenario where there are multiple cases to be performed for different conditions.

```
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
```

```c
}

#include<stdio.h>
int main(){
int number=0;
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");
}
return 0;
}
```

C Switch Statement

The switch statement in C is an alternative to if-else-if ladder statement which allows us to execute multiple operations for the different possible values of a single variable called switch

variable. Here, We can define various statements in the multiple cases for the different values of a single variable.

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

1. The *switch expression* must be of an integer or character type.

2. The *case value* must be an integer or character constant.

3. The *case value* can be used only inside the switch statement.

4. The *break statement* in the switch case is not a must. It is optional. If there is no break statement found in the case, all the cases will be executed present after the matched case. It is known as *fall through* the state of C switch statement.

11. LOOPING STATEMENTS

The looping can be defined as repeating the same process multiple times until a specific condition satisfies. There are three types of loops used in the C language.

Advantage of loops in C

1) It provides code reusability.

2) Using loops, we do not need to write the same code again and again.

3) Using loops, we can traverse over the elements of data structures (array or linked lists).

Types of C Loops

There are three types of loops in C language that is given below:

- do while
- while
- For

do-while loop in C

- The do-while loop continues until a given condition satisfies. It is also called post tested loop. It is used when it is necessary to execute the loop at least once (mostly menu driven programs).

  Syntax :

  do{

  //code to be executed printf("hi");

  }while(condition); a<10

  Example :

  ```c
  #include <stdio.h>
  int main() {
  int i = 1;
  ```

```c
do {
printf("%d\n", i);
i++;
} while (i<= 5); 1<=5
return 0;
}
```

while loop in C

The while loop in c is to be used in the scenario where we don't know the number of iterations in advance. The block of statements is executed in the while loop until the condition specified in the while loop is satisfied. It is also called a pre-tested loop.

Syntax :

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

Example :

```c
#include<stdio.h>
int main(){
int i=1;
while(i<=10){
printf("%d \n",i);
i++;
}
```

```
return 0;

}
```

For loop in C

The for loop is used in the case where we need to execute some part of the code until the given condition is satisfied. The for loop is also called as a pre-tested loop. It is better to use for loop if the number of iteration is known in advance.

Syntax :

for(initialization;condition;incr/decr)   for(i=10;i<=10;i++)

{

//code to be executed

}

Example :

```c
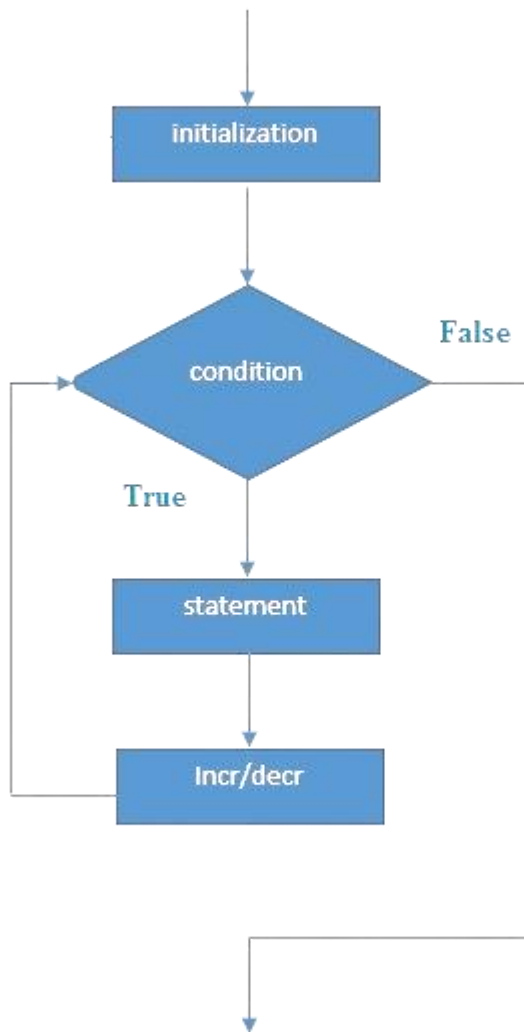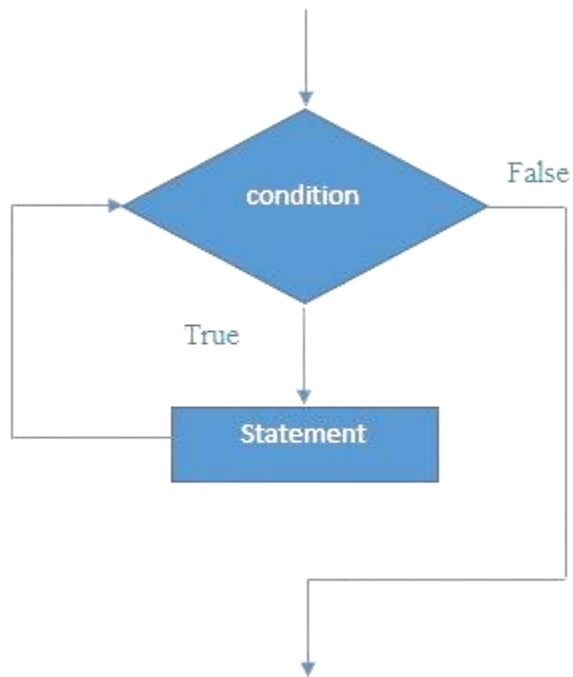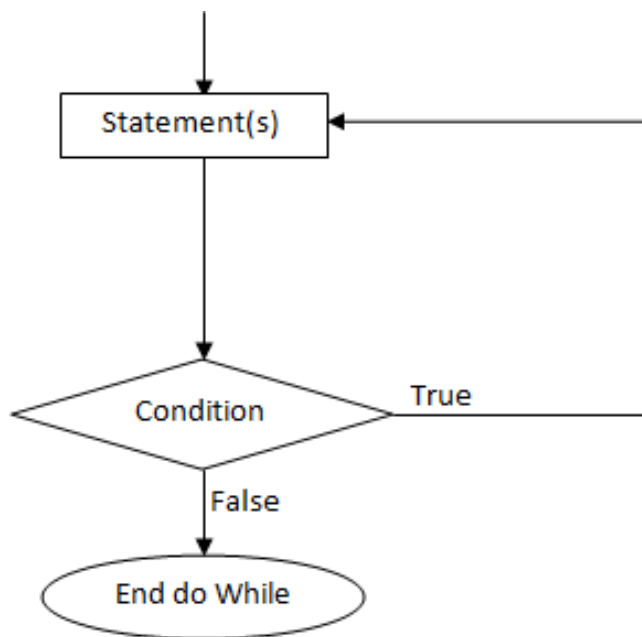#include<stdio.h>
int main(){
int i=0;
for(i=1;i<=10;i++){
printf("%d \n",i);
}
return 0;
}
```

For -Loop

While Loop

Do-while loop

Nested Loops in C

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

Syntax of Nested loop

Outer_loop

{

  Inner_loop

  {

    // inner loop statements.

  }

    // outer loop statements.

}

Example :

#include <stdio.h>

int main()

{

  int n;// variable declaration

  printf("Enter the value of n :");

  scanf("%d",&n);

  // Displaying the n tables.

```
    for(int i=1;i<=n;i++)  // outer loop ——---rows

  {

    for(int j=1;j<=10;j++)  // inner loop -------- columns

   {

      printf("%d\t",(i*j)); // printing the value.

   }

    printf("\n");

  }
```

Explanation of the above code

- ○ First, the 'i' variable is initialized to 1 and then program control passes to the i<=n.

- ○ The program checks whether the condition 'i<=n' is true or not.

- ○ If the condition is true, then the program control passes to the inner loop.

- ○ The inner loop will get executed until the condition is true.

- ○ After the execution of the inner loop, the control moves back to the update of the outer loop, i.e., i++.

- ○ After incrementing the value of the loop counter, the condition is checked again, i.e., i<=n.

- ○ If the condition is true, then the inner loop will be executed again.

- ○ This process will continue until the condition of the outer loop is true.

Nested while loop

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

       while(condition)

```
    {

        while(condition)

        {

            // inner loop statements.

        }

    // outer loop statements.

    }
```

INFINITE LOOP

An infinite loop is a looping construct that does not terminate the loop and executes the loop forever. It is also called an indefinite loop or an endless loop.

Infinite for loop

```
    for(; ;)
{

    // body of the for loop.

}
```

Infinite while loop

```
    while(1)
    {
        // body of the loop..
```

}

Infinite do while loop

```
do
{
    // body of the loop..
}while(1);
```

Example :

```c
#include <stdio.h>
int main()
{
  int i=0;
  while(1)
  {
     i++;
     printf("i is :%d",i);
  }
return 0;
}
```

12. FUNCTIONS

C Functions

In c, we can divide a large program into the basic building blocks known as function. The function contains the set of programming statements enclosed by {}. A function can be called multiple times to provide reusability and modularity to the C program. In other words, we can say that the collection of functions creates a program. The function is also known as *procedure or subroutine in* other programming languages.

Advantage of functions in C

There are the following advantages of C functions.

- By using functions, we can avoid rewriting the same logic/code again and again in a program.
- We can call C functions any number of times in a program and from any place in a program.
- We can track a large C program easily when it is divided into multiple functions.
- Reusability is the main achievement of C functions.
- However, Function calling is always overhead in a C program.

Function Aspects

There are three aspects of a C function.

- **Function declaration-** A function must be declared globally in a c program to tell the compiler about the function name, function parameters, and return type.
  Function call -Function can be called from anywhere in the program. The parameter list must not differ in function calling and function declaration. We must pass the same number of functions as it is declared in the function declaration.

○ **Function definition-** It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is called. Here, we must notice that only one value can be returned from the function.

1	Function declaration	return_type function_name (argument list);
2	Function call	function_name (argument_list)
3	Function definition	return_type function_name (argument list) {function body;}

The syntax of creating function in c language is given below:

1. return_type function_name(data_type parameter...){
2. //code to be executed
3. }

Types of Functions

There are two types of functions in C programming:

1. Library Functions: are the functions which are declared in the C header files such as scanf(), printf(), gets(), puts(), ceil(), floor() etc.
2. User-defined functions: are the functions which are created by the C programmer, so that he/she can use it many times. It reduces the complexity of a big program and optimizes the code.
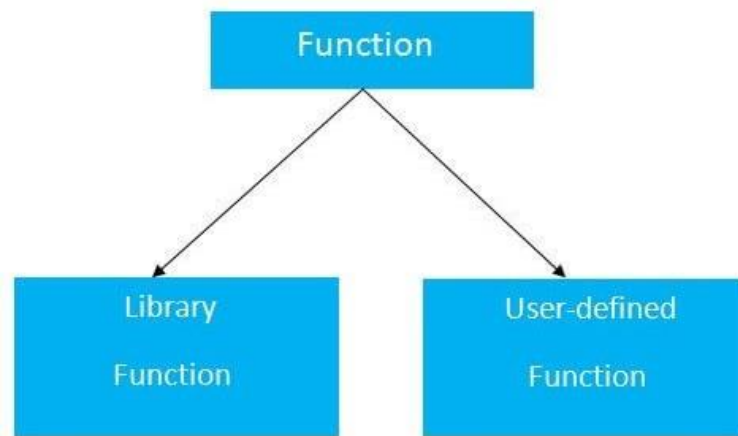
Return Value

A C function may or may not return a value from the function. If you don't have to return any value from the function, use void for the return type.

Let's see a simple example of C function that doesn't return any value from the function.

Example without return value:

1. void hello() —function name
2. {
3. printf("hello c");
4. }

If you want to return any value from the function, you need to use any data type such as int, long, char, etc. The return type depends on the value to be returned from the function.

Let's see a simple example of a C function that returns int value from the function.

Example with return value:

1. int get(){
2. return 10;   o/p : 10

3. }

In the above example, we have to return 10 as a value, so the return type is int. If you want to return floating-point value (e.g., 10.2, 3.1, 54.5, etc), you need to use float as the return type of the method.

1. float get(){
2. return 10.2;
3. }

Now, you need to call the function, to get the value of the function.

Different aspects of function calling

A function may or may not accept any argument. It may or may not return any value. Based on these facts, There are four different aspects of function calls.

- function **without arguments** and **without return value**
- function **without arguments** and **with return value**
- function **with arguments** and **without return value**
- function **with arguments** and **with return value**

Example for Function without argument and return value

Example 1

1. #include<stdio.h>
2. void printName(); —function declaration
3. void main ()
4. {
5. printf("Hello ");
6. printName(); —calling function

7. }
8. void printName()
9. {
10.   printf("Hi");
11. }

Output

Hello Javatpoint

Example for Function without argument and with return value

Example 1

1. #include<stdio.h>
2. int sum();
3.   void main()
4. {
5.   int result;
6.   printf("\nGoing to calculate the sum of two numbers:");
7.   result = sum();
8.   printf("%d",result);
9. }
10. int sum()
11. {
12.   int a,b;
13.   printf("\nEnter two numbers");
14.   scanf("%d %d",&a,&b);
15.   return a+b;
16. }

Output

Going to calculate the sum of two numbers:

Enter two numbers 10

24

The sum is 34

Example for Function with argument and without return value

Example 1

1. #include<stdio.h>
2. void sum(int, int); —-function decl
3. void main()
4. {
5.   int a,b,result;
6.   printf("\nGoing to calculate the sum of two numbers:");
7.   printf("\nEnter two numbers:");
8.   scanf("%d %d",&a,&b);
9.   sum(a,b); —--->calling fuction
10. }
11. void sum(int a, int b)
12. {
13.   printf("\nThe sum is %d",a+b);
14. }

Output

Going to calculate the sum of two numbers:

Enter two numbers 10

24

The sum is 34

Example for Function with argument and with return value

Example 1

1. #include<stdio.h>
2. int sum(int, int);
3. void main()
4. {
5.   int a,b,result;
6.   printf("\nGoing to calculate the sum of two numbers:");
7.   printf("\nEnter two numbers:");
8.   scanf("%d %d",&a,&b);
9.   result = sum(a,b);
10.   printf("\nThe sum is : %d",result);
11. }
12. int sum(int a, int b)
13. {
14.   return a+b;
15. }

Output

Going to calculate the sum of two numbers:

Enter two numbers:10

20

The sum is : 30

C Library Functions

Library functions are the inbuilt function in C that are grouped and placed at a common place called the library. Such functions are used to perform some specific operations. For example, printf is a library function used to print on the console. The library functions are created by the designers of compilers. All C standard library functions are defined inside the different header files saved with the extension .h. We need to include these header files in our program to make use of the library functions defined in such header files. For example, To use the library functions such as printf/scanf we need to include stdio.h in our program which is a header file that contains all the library functions regarding standard input/output.

The list of mostly used header files is given in the following table.

S.NO	HEADER FILE	DESCRIPTION
1	stdio.h	This is a standard input/output header file. It contains all the library functions regarding standard input/output.
2	conio.h	This is a console input/output header file.
3	string.h	It contains all string related library functions like gets(), puts(),etc.
4	stdlib.h	This header file contains all the general library functions like malloc(), calloc(), exit(), etc.
5	math.h	This header file contains all the math operations related

		functions like sqrt(), pow(), etc.
6	time.h	This header file contains all the time-related functions.
7	ctype.h	This header file contains all character handling functions.
8	stdarg.h	Variable argument functions are defined in this header file.
9	signal.h	All the signal handling functions are defined in this header file.
10	setjmp.h	This file contains all the jump functions.
11	locale.h	This file contains locale functions.
12	errno.h	This file contains error handling functions.
13	assert.h	This file contains diagnostics functions.

12. STORAGE CLASSES

Storage classes in C are used to determine the lifetime, visibility, memory location, and initial value of a variable. There are four types of storage classes in C

- ○ Automatic
- ○ External
- ○ Static
- ○ Register

Storage Classes	Storage Place	Default Value	Scope	Lifetime
auto	RAM	Garbage Value	Local	Within function
extern	RAM	Zero	Global	Till the end of the main program May Be declared anywhere in the program
static	RAM	Zero	Local	Till the end of the main program, Retains value between multiple functions call
register	Register	Garbage Value	Local	Within the function

Automatic

- ○ Automatic variables are allocated memory automatically at runtime.
- ○ The visibility of the automatic variables is limited to the block in which they are defined.

- The scope of the automatic variables is limited to the block in which they are defined. The automatic variables are initialized to garbage by default.
- The memory assigned to automatic variables gets freed upon exiting from the block.
- The keyword used for defining automatic variables is auto.
- Every local variable is automatic in C by default.

Example 1

1. #include <stdio.h>
2. int main()
3. {
4. auto int a; //auto
5. char b;
6. float c;
7. printf("%d %c %f",a,b,c); // printing initial default value of automatic variables a, b, and c.
8. return 0;]
9. }

Output:

garbage garbage garbage

Example 2

1. #include <stdio.h>
2. int main()
3. {
4. int a = 10,i;
5. printf("%d ",++a);
6. {

7. int a = 20;

8. for (i=0;i<3;i++)

9. {

10. printf("%d ",a); // 20 will be printed 3 times since it is the local value of a

11. }

12. }

13. printf("%d ",a); // 11 will be printed since the scope of a = 20 is ended.

14. }

Output:

 11 20 20 20 11

Static

- ○ The variables defined as static specifiers can hold their value between the multiple function calls.

- ○ Static local variables are visible only to the function or the block in which they are defined.

- ○ The same static variable can be declared many times but can be assigned at only one time.

- ○ Default initial value of the static integral variable is 0 otherwise null.

- ○ The visibility of the static global variable is limited to the file in which it has been declared.

- ○ The keyword used to define a static variable is static.

Example 1

1. #include<stdio.h>

2. static char c;

3. static int i;

4. static float f;

5. static char s[100];

6. void main ()

7. {

8. printf("%c %d %f %s",c,i,f,s); // the initial default value of c, i, and f will be printed.

9. }

Output:

0 0 0.000000 (null)

Example 2

1. #include<stdio.h>

2. void sum()

3. {

4. static int a = 10;

5. static int b = 24;

6. printf("%d %d \n",a,b);

7. a++;

8. b++;

9. }

10. void main()

11. {

12. int i;

13. for(i = 0; i< 3; i++)

14. {

15. sum(); // The static variables hold their value between multiple function calls.

16. }

17. }

Output:

10 24

11 25

12 26

Register

○ The variables defined as the register are allocated the memory into the CPU registers depending upon the size of the memory remaining in the CPU.

○ We can not dereference the register variables, i.e., we can not use & operator for the register variable.

○ The access time of the register variables is faster than the automatic variables.

○ The initial default value of the register local variables is 0.

○ The register keyword is used for the variable which should be stored in the CPU register. However, it is the compiler's choice whether or not; the variables can be stored in the register.

○ We can store pointers into the register, i.e., a register can store the address of a variable.

○ Static variables can not be stored into the register since we can not use more than one storage specifier for the same variable.

Example 1

1. #include <stdio.h>
2. int main()
3. {
4. register int a; // variable a is allocated memory in the CPU register. The initial default value of a is 0.
5. printf("%d",a);

6. }

Output:

0

Example 2

1. #include <stdio.h>
2. int main()
3. {
4. register int a = 0;
5. printf("%u",&a); // This will give a compile time error since we can not access the address of a register variable.
6. }

Output:

main.c:5:5: error: address of register variable ?a? requested

printf("%u",&a);

External

○ The external storage class is used to tell the compiler that the variable defined as extern is declared with an external linkage elsewhere in the program.

○ The variables declared as extern are not allocated any memory. It is only a declaration and intended to specify that the variable is declared elsewhere in the program.

○ The default initial value of the external integral type is 0 otherwise null.

○ We can only initialize the external variable globally, i.e., we can not initialize the external variable within any block or method.

○ An external variable can be declared many times but can be initialized at only once.

- If a variable is declared as external then the compiler searches for that variable to be initialized somewhere in the program which may be extern or static. If it is not, then the compiler will show an error.

Example 1

1. #include <stdio.h>
2. int main()
3. {
4. extern int a;
5. printf("%d",a);
6. }

Output

main.c:(.text+0x6): undefined reference to `a'

collect2: error: ld returned 1 exit status

Example 2

1. #include <stdio.h>
2. int a;
3. int main()
4. {
5. extern int a; // variable a is defined globally, the memory will not be allocated to a
6. printf("%d",a);
7. }

Output

0

Example 3

1. #include <stdio.h>
2. int a;
3. int main()
4. {
5. extern int a = 0; // this will show a compiler error since we can not use extern and initializer at same time
6. printf("%d",a);
7. }

Output

compile time error

main.c: In function ?main?:

main.c:5:16: error: ?a? has both ?extern? and initializer

extern int a = 0;

Example 4

1. #include <stdio.h>
2. int main()
3. {
4. extern int a; // Compiler will search here for a variable a defined and initialized somewhere in the pogram or not.
5. printf("%d",a);
6. }
7. int a = 20;

Output

Example 5

1. extern int a;
2. int a = 10;
3. #include <stdio.h>
4. int main()
5. {
6. printf("%d",a);
7. }
8. int a = 20; // compiler will show an error at this line

Output

Compile Error

13. ARRAY CONCEPTS

An array is defined as the collection of similar types of data items stored at contiguous memory locations. Arrays are the derived data type in C programming language which can store the

primitive type of data such as int, char, double, float, etc. It also has the capability to store the collection of derived data types, such as pointers, structure, etc. The array is the simplest data structure where each data element can be randomly accessed by using its index number.

Properties of Array

The array contains the following properties.

- Each element of an array is of the same data type and carries the same size, i.e., int = 4 bytes.
- Elements of the array are stored at contiguous memory locations where the first element is stored at the smallest memory location.
- Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

Advantage of C Array

1) Code Optimization: Less code to the access the data.

2) Ease of traversing: By using the for loop, we can retrieve the elements of an array easily.

3) Ease of sorting: To sort the elements of the array, we need a few lines of code only.

4) Random Access: We can access any element randomly using the array.

Disadvantage of C Array

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

Declaration of C Array

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

1. data_type array_name[array_size];

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

1. int marks[5]; int marks[5]={80,60,70,85}

Here, int is the *data_type*, marks are the *array_name*, and 5 is the *array_size*.

Initialization of C Array

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

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

80	60	70	85	75
marks[0]	marks[1]	marks[2]	marks[3]	marks[4]

Initialization of Array

C array example

1. #include<stdio.h>
2. int main(){
3. int i=0;

4. int marks[5];//declaration of array   int marks[5]={80,60,70,85,75};

5. marks[0]=80;//initialization of array

6. marks[1]=60;

7. marks[2]=70;

8. marks[3]=85;

9. marks[4]=75;

10. //traversal of array

11. for(i=0;i<5;i++){

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

13. }//end of for loop

14. return 0;

15. }

Output

80

60

70

85

75

1. #include<stdio.h>

2. void main ()

3. {

4.   int i, j,temp;

5.   int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};

6.   for(i = 0; i<10; i++) i=0

7.   {

8.   **for**(j = i+1; j<10; j++)   j=1 j=2 j=3 j=9

9. {

10.   **if**(a[j] > a[i]) a[1]>a[0] 9>10 a[2]>a[0] 7>10 101>10 a[3]>a[0]

11.   {

12.   temp = a[i]; temp= 7; temp = 10

13.   a[i] = a[j];   a[i]= 101; a[i]= 101

14.   a[j] = temp;   a[j]=7; a[j]=10

15.   }

16.   }

17. }

18. printf(**"Printing Sorted Element List ...\n"**);

19. **for**(i = 0; i<10; i++)

20. {

21.   printf(**"%d\n"**,a[i]);

22. }

23. }

Two Dimensional Array in C

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

Declaration of two dimensional Array in C

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

1. data_type array_name[rows][columns];

Consider the following example.

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

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

Initialization of 2D Array in C

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

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

Two-dimensional array example in C

1. #include<stdio.h>
2. **int** main(){
3. **int** arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
4. //traversing 2D array
5. **for**(i=0;i<4;i++)   //i=0,1,2,3
6. {

7. **for**(j=0;j<3;j++){   j=0,1,2
8.   printf("arr[%d] [%d] = %d \n",i,j,arr[i][j]);
9. }//end of j

10. }//end of i
11. **return** 0;
12. }

Output

arr[0][0] = 1

arr[0][1] = 2

arr[0][2] = 3

arr[1][0] = 2

arr[1][1] = 3

arr[1][2] = 4

arr[2][0] = 3

arr[2][1] = 4

arr[2][2] = 5

arr[3][0] = 4

arr[3][1] = 5

arr[3][2] = 6

C 2D array example: Storing elements in a matrix and printing it.

1. #include <stdio.h>
2. **void** main ()

```
3.  {
4.      int arr[3][3],i,j;
5.      for (i=0;i<3;i++)
6.      {
7.          for (j=0;j<3;j++)
8.          {
9.              printf("Enter a[%d][%d]: ",i,j);
10.             scanf("%d",&arr[i][j]);
11.         }
12.     }
13.     printf("\n printing the elements .... \n");
14.     for(i=0;i<3;i++)
15.     {
16.         printf("\n");
17.         for (j=0;j<3;j++)
18.         {
19.             printf("%d\t",arr[i][j]);
20.         }
21.     }
```

Enter a[0][0]: 56

Enter a[0][1]: 10

Enter a[0][2]: 30

Enter a[1][0]: 34

Enter a[1][1]: 21

Enter a[1][2]: 34

Enter a[2][0]: 45

Enter a[2][1]: 56

Enter a[2][2]: 78

printing the elements ....

56   10   30

34   21   34

45   56   78

Passing array to a function

1. #include <stdio.h>
2. **void** getarray(**int** arr[])
3. {
4.   printf("Elements of array are : ");
5.   **for**(**int** i=0;i<5;i++)
6.   {
7.   printf("%d ", arr[i]);
8.   }
9. }
10. **int** main()
11. {
12.   **int** arr[5]={45,67,34,78,90};

13.   getarray(arr);
14.   **return** 0;
15. }

In the above program, we have first created the array **arr[]** and then we pass this array to the function getarray(). The **getarray()** function prints all the elements of the array arr[].

Output

```
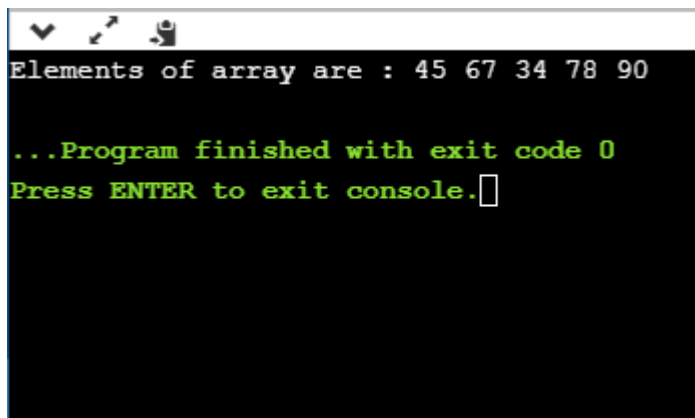Elements of array are : 45 67 34 78 90

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

Passing Array to Function in C

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

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

1.   functionname(arrayname);//passing array

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

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

First way:

1. return_type function(type arrayname[])

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

Second way:

1. return_type function(type arrayname[SIZE])

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

Third way:

1. return_type function(type *arrayname)

You can also use the concept of a pointer. In pointer chapter, we will learn about it.

C language passing an array to function example

1. #include<stdio.h>
2. int minarray(int arr[],int size){
3. int min=arr[0];
4. int i=0;
5. for(i=1;i<size;i++){
6. if(min>arr[i]){
7. min=arr[i];
8. }
9. }//end of for
10. return min;
11. }//end of function

12.

13. int main(){

14. int i=0,min=0;

15. int numbers[]={4,5,7,3,8,9};//declaration of array

16.

17. min=minarray(numbers,6);//passing array with size

18. printf("minimum number is %d \n",min);

19. return 0;

20. }

Output

Minimum number is 3

Returning array from the function

As we know, a function can not return more than one value. However, if we try to write the return statement as return a, b, c; to return three values (a,b,c), the function will return the last mentioned value which is c in our case. In some problems, we may need to return multiple values from a function. In such cases, an array is returned from the function.

Returning an array is similar to passing the array into the function. The name of the array is returned from the function. To make a function returning an array, the following syntax is used.

1. **int** * Function_name() {

2. //some statements;

3. **return** array_type;

4. }

To store the array returned from the function, we can define a pointer which points to that array. We can traverse the array by increasing that pointer since pointer initially points to the base

address of the array. Consider the following example that contains a function returning the sorted array.

1. #include<stdio.h>
2. **int*** Bubble_Sort(**int[]**);
3. **void** main ()
4. {
5.   **int** arr[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
6.   **int** *p = Bubble_Sort(arr), i;
7.   printf("printing sorted elements ...\n");
8.   **for**(i=0;i<10;i++)
9.   {
10.   printf("%d\n",*(p+i));
11.   }
12. }
13. **int*** Bubble_Sort(**int** a[]) //array a[] points to arr.
14. {
15. **int** i, j,temp;
16.   **for**(i = 0; i<10; i++)
17.   {
18.   **for**(j = i+1; j<10; j++)
19.   {
20.   **if**(a[j] < a[i])
21.   {
22.   temp = a[i];
23.   a[i] = a[j];
24.   a[j] = temp;
25.   }
26.   }
27.   }

28.   **return** a;

29. }

Output

Printing Sorted Element List ...

7

9

10

12

23

23

34

44

78

101

14. STRUCTURE

Structure in c is a user-defined data type that enables us to store the collection of different data types. Each element of a structure is called a member. Structures ca; simulate the use of classes and templates as it can store various information

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

1. struct structure_name
2. {
3.   data_type member1;
4.   data_type member2;
5.   .
6.   .
7.   data_type memeberN;
8. };

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

1. struct employee
2. {   int id;
3.   char name[20];
4.   float salary;
5. };

The following image shows the memory allocation of the structure employee that is defined in the above example.

1000	1004		1014	1018

int id   char Name[10]   float salary

struct Employee   sizeof (emp) = 4 + 10 + 4 = 18 bytes
{
    int id;   where;
    char Name[10];   sizeof (int) = 4 byte   ☐ ──→ 1 byte
    float salary;   sizeof (char) = 1 byte
} emp;   sizeof (float) = 4 byte

Here, struct is the keyword; employee is the name of the structure; id, name, and salary are the members or fields of the structure. Let's understand it by the diagram given below:

```
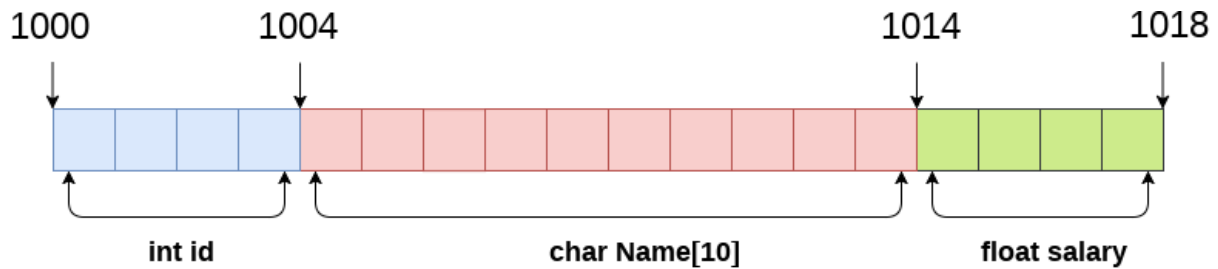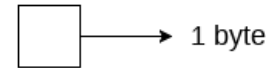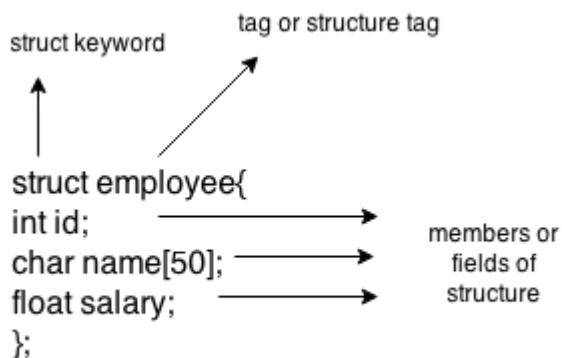            tag or structure tag
struct keyword
     ↑         ↗
     │        ╱
struct employee{
int id;          ────────→
char name[50];   ────────→    members or
float salary;    ────────→    fields of
};                            structure
```

JavaTpoint.com

Declaring structure variable

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

1. By struct keyword within main() function

2. By declaring a variable at the time of defining the structure.

1st way:

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

1. struct employee   struct employee id,name[50],salary;
2. {
3. int id;
4.   char name[50];
5.   float salary;
6. };

Now write given code inside the main() function.

1. struct employee e1, e2; —->declaring structure

The variables e1 and e2 can be used to access the values stored in the structure. Here, e1 and e2 can be treated in the same way as the objects in C++ and Java.

2nd way:

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

1. struct employee
2. { int id;
3.   char name[50];   e1.id   e1.salary
4.   float salary;
5. }e1,e2;

Accessing members of the structure

There are two ways to access structure members:

1. By . (member or dot operator)

2. By -> (structure pointer operator)

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

   e1.id

C Structure example

Let's see a simple example of structure in C language.

1. #include<stdio.h>

2. #include <string.h>

3. **struct** employee

4. {   **int** id;

5.   **char** name[50];

6. }e1; //declaring e1 variable for structure

7. **int** main( )

8. {

9.   //store first employee information

10.   e1.id=101;

11.   strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array

12.   //printing first employee information

13.   printf("employee 1 id : %d\n", e1.id);

14.   printf("employee 1 name : %s\n", e1.name);

15. **return** 0;

16. }

Output:

   employee 1 id : 101

   employee 1 name : Sonoo Jaiswal

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

2.  #include <string.h>

3.  struct employee

4.  {   int id;

5.      char name[50];

6.      float salary;

7.  }e1,e2;  //declaring e1 and e2 variables for structure

8.  int main( )

9.  {

10.  //store first employee information

11.  e1.id=101;

12.  strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array

13.  e1.salary=56000;

14.

15.  //store second employee information

16.  e2.id=102;

17.  strcpy(e2.name, "James Bond");

18.  e2.salary=126000;

19.

20.  //printing first employee information

21.  printf( "employee 1 id : %d\n", e1.id);

22.  printf( "employee 1 name : %s\n", e1.name);

23.  printf( "employee 1 salary : %f\n", e1.salary);

24.

25.  //printing second employee information

26.  printf( "employee 2 id : %d\n", e2.id);

27.  printf( "employee 2 name : %s\n", e2.name);

28.  printf( "employee 2 salary : %f\n", e2.salary);

29.  return 0;
```

30. }

Output:

employee 1 id : 101

employee 1 name : Sonoo Jaiswal

employee 1 salary : 56000.000000

employee 2 id : 102

employee 2 name : James Bond

employee 2 salary : 126000.000000

ARRAY OF STRUCTURES :

1. #include<stdio.h>
2. #include <string.h>
3. **struct** student{
4. **int** rollno;
5. **char** name[10];
6. };
7. **int** main(){
8. **int** i;
9. **struct** student st[5];
10. printf("Enter Records of 5 students");
11. **for**(i=0;i<5;i++){
12. printf("\nEnter Rollno:");
13. scanf("%d",&st[i].rollno);
14. printf("\nEnter Name:");

15. scanf("%s",&st[i].name);

16. }

17. printf("\nStudent Information List:");

18. **for**(i=0;i<5;i++){

19. printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);

20. }

21.   **return** 0;

22. }

Output:

Enter Records of 5 students

Enter Rollno:1

Enter Name:Sonoo

Enter Rollno:2

Enter Name:Ratan

Enter Rollno:3

Enter Name:Vimal

Enter Rollno:4

Enter Name:James

Enter Rollno:5

Enter Name:Sarfraz

Student Information List:

Rollno:1, Name:Sonoo

Rollno:2,   Name:Ratan

Rollno:3,   Name:Vimal

Rollno:4,   Name:James

Rollno:5, Name:Sarfraz

Nested Structure in C

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

1. #include<stdio.h>
2. **struct** address
3. {
4.   **char** city[20];
5.   **int** pin;
6.   **char** phone[14];
7. };
8. **struct** employee
9. {
10.   **char** name[20];
11.   **struct** address add; emp.add.pin , emp.add.city

```
12. };
13. void main ()
14. {
15.     struct employee emp;
16.     printf("Enter employee information?\n");
17.     scanf("%s %s %d %s",&emp.name,&emp.add.city, &emp.add.pin, &emp.add.phone);
18.     printf("Printing the employee information....\n");
19.     printf("name: %s\nCity: %s\nPincode: %d\nPhone:
        %s",emp.name,emp.add.city,emp.add.pin,emp.add.phone);
20. }
```

Output

Enter employee information?

Arun

Delhi

110001

1234567890

Printing the employee information....

name: Arun

City: Delhi

Pincode: 110001

Phone: 1234567890

The structure can be nested in the following ways.

1. By separate structure

2. By Embedded structure

1) Separate structure

Here, we create two structures, but the dependent structure should be used inside the main structure as a member. Consider the following example.

1. **struct** Date
2. {
3.   **int** dd;
4.   **int** mm;
5.   **int** yyyy;
6. };
7. **struct** Employee
8. {
9.   **int** id;
10.   **char** name[20];
11.   **struct** Date doj;
12. }emp1;

As you can see, doj (date of joining) is the variable of type Date. Here doj is used as a member in Employee structure. In this way, we can use Date structure in many structures.

2) Embedded structure

The embedded structure enables us to declare the structure inside the structure. Hence, it requires less line of codes but it can not be used in multiple data structures. Consider the following example.

1. **struct** Employee
2. {
3.   **int** id;
4.   **char** name[20];
5.   **struct** Date
6.   {
7.   **int** dd;
8.   **int** mm;
9.   **int** yyyy;
10.   }doj;
11. }emp1;

Accessing Nested Structure

We can access the member of the nested structure by Outer_Structure.Nested_Structure.member as given below:

1. e1.doj.dd
2. e1.doj.mm
3. e1.doj.yyyy

C Nested Structure example

Let's see a simple example of the nested structure in C language.

1. #include <stdio.h>
2. #include <string.h>
3. **struct** Employee
4. {
5.   **int** id;

```c
6.    char name[20];
7.    struct Date
8.    {
9.      int dd;
10.     int mm;
11.     int yyyy;
12.   }doj;
13. }e1;
14. int main( )
15. {
16.   //storing employee information
17.   e1.id=101;
18.   strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
19.   e1.doj.dd=10;
20.   e1.doj.mm=11;
21.   e1.doj.yyyy=2014;
22.
23.   //printing first employee information
24.   printf( "employee id : %d\n", e1.id);
25.   printf( "employee name : %s\n", e1.name);
26.   printf( "employee date of joining (dd/mm/yyyy) : %d/%d/%d\n",
      e1.doj.dd,e1.doj.mm,e1.doj.yyyy);
27.   return 0;
28. }
```

15.UNIONS

Union can be defined as a user-defined data type which is a collection of different variables of different data types in the same memory location. The union can also be defined as many members, but only one member can contain a value at a particular point in time.

Union is a user-defined data type, but unlike structures, they share the same memory location.

Let's understand this through an example.

1. **struct** abc
2. {
3.   **int** a; - 4bytes
4.   **char** b; -1byte
5. }

The above code is the user-defined structure that consists of two members, i.e., 'a' of type **int** and 'b' of type **character**. When we checked the addresses of 'a' and 'b', we found that their addresses are different. Therefore, we conclude that the members in the structure do not share the same memory location.

When we define the union, then we found that union is defined in the same way as the structure is defined but the difference is that union keyword is used for defining the union data type, whereas the struct keyword is used for defining the structure. The union contains the data members, i.e., 'a' and 'b', when we check the addresses of both the variables then we find that both have the same addresses. It means that the union members share the same memory location.

Let's have a look at the pictorial representation of the memory allocation.

The below figure shows the pictorial representation of the structure. The structure has two members; i.e., one is of integer type, and the another one is of character type. Since 1 block is

equal to 1 byte; therefore, 'a' variable will be allocated 4 blocks of memory while 'b' variable will be allocated 1 block of memory.

The below figure shows the pictorial representation of union members. Both the variables are sharing the same memory location and having the same initial address.

In union, members will share the memory location. If we try to make changes in any of the member then it will be reflected to the other member as well. Let's understand this concept through an example.

1. **union** abc
2. {
3.   **int** a;
4. **char** b;
5. }var;
6. **int** main()
7. {
8.   var.a = 66;
9.   printf("\n a = %d", var.a);
10. printf("\n b = %d", var.b);
11. }

In the above code, union has two members, i.e., 'a' and 'b'. The 'var' is a variable of union abc type. In the **main()** method, we assign the 66 to 'a' variable, so var.a will print 66 on the screen. Since both 'a' and 'b' share the memory location, **var.b** will print **'B'** (ascii code of 66).

Deciding the size of the union

The size of the union is based on the size of the largest member of the union.

Let's understand through an example.

1. **union** abc{
2. **int** a;
3. **char** b;
4. **float** c;
5. **double** d;
6. };
7. **int** main()
8. {
9. printf("Size of union abc is %d", **sizeof**(**union** abc));
10. **return** 0;
11. }

As we know, the size of int is 4 bytes, size of char is 1 byte, size of float is 4 bytes, and the size of double is 8 bytes. Since the double variable occupies the largest memory among all the four variables, so total 8 bytes will be allocated in the memory. Therefore, the output of the above program would be 8 bytes.

Accessing members of union using pointers

We can access the members of the union through pointers by using the (->) arrow operator.

Let's understand through an example.

1. #include <stdio.h>
2. **union** abc
3. {
4. **int** a;
5. **char** b;
6. };
7. **int** main()
8. {

9.   **union** abc *ptr; // pointer variable declaration

10.   **union** abc var;

11.   var.a= 90;

12.   ptr = &var;

13.   printf("The value of a is : %d", ptr->a);

14.   **return** 0;

15. }

In the above code, we have created a pointer variable, i.e., *ptr, that stores the address of var variable. Now, ptr can access the variable 'a' by using the (->) operator. Hence the output of the above code would be 90.

Why do we need C unions?

Consider one example to understand the need for C unions. Let's consider a store that has two items:

- ○ Books

- ○ Shirts

Store owners want to store the records of the above-mentioned two items along with the relevant information. For example, Books include Title, Author, no of pages, price, and Shirts include Color, design, size, and price. The 'price' property is common in both items. The Store owner wants to store the properties, then how he/she will store the records.

Initially, they decided to store the records in a structure as shown below:

1. **struct** store

2. {

3.   **double** price;

4.   **char** *title;

5.   **char** *author;

6.   **int** number_pages;

7.   **int** color;

8.   **int** size;

9.   **char** *design;

10. };

The above structure consists of all the items that store owner wants to store. The above structure is completely usable but the price is common property in both the items and the rest of the items are individual. The properties like price, *title, *author, and number_pages belong to Books while color, size, *design belongs to Shirt.

Let's see how can we access the members of the structure.

1. **int** main()

2. {

3.   **struct** store book;

4.   book.title = "C programming";

5. book.author = "Paulo Cohelo";

6. book.number_pages = 190;

7. book.price = 205;

8. printf("Size is : %ld bytes", **sizeof**(book));

9. **return** 0;

10. }

In the above code, we have created a variable of type **store**. We have assigned the values to the variables, title, author, number_pages, price but the book variable does not possess the properties such as size, color, and design. Hence, it's a wastage of memory. The size of the above structure would be 44 bytes.

We can save lots of space if we use unions.

1. #include <stdio.h>

```c
2.  struct store
3.  {
4.     double price;
5.     union
6.     {
7.        struct{
8.        char *title;
9.        char *author;
10.       int number_pages;
11.       } book;
12.
13.       struct {
14.       int color;
15.       int size;
16.       char *design;
17.       } shirt;
18.    }item;
19. };
20. int main()
21. {
22.    struct store s;
23.    s.item.book.title = "C programming";
24.    s.item.book.author = "John";
25.    s.item.book.number_pages = 189;
26.    printf("Size is %ld", sizeof(s));
27.    return 0;
28. }
```

In the above code, we have created a variable of type store. Since we used the unions in the above code, the largest memory occupied by the variable would be considered for the memory

allocation. The output of the above program is 32 bytes. In the case of structures, we obtained 44 bytes, while in the case of unions, the size obtained is 44 bytes. Hence, 44 bytes is greater than 32 bytes saving lots of memory space.

C Pointers

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte.

Consider the following example to define a pointer which stores the address of an integer.

1. **int** n = 10;
2. **int*** p = &n; // Variable p of type pointer is pointing to the address of the variable n of type integer.

Declaring a pointer

The pointer in c language can be declared using * (asterisk symbol). It is also known as indirection pointer used to dereference a pointer.

1. **int** *a;//pointer to int
2. **char** *c;//pointer to char

Pointer Example

An example of using pointers to print the address and value is given below.

aaa3 fff4

fff4 50

p number
(pointer) (normal variable)
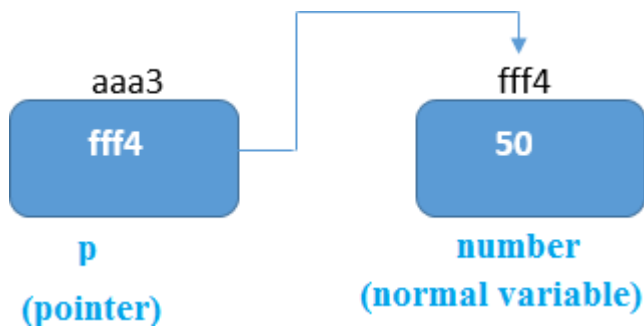
As you can see in the above figure, the pointer variable stores the address of the number variable, i.e., fff4. The value of the number variable is 50. But the address of the pointer variable p is aaa3.

By the help of * (**indirection operator**), we can print the value of pointer variable p.

Let's see the pointer example as explained for the above figure.

1. #include<stdio.h>
2. **int** main(){
3. **int** number=50;
4. **int** *p;
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %x \n",p); // p contains the address of the number therefore printing p gives the address of number.
7. printf("Value of p variable is %d \n",*p); // As we know that * is used to dereference a pointer therefore if we print *p, we will get the value stored at the address contained by p.
8. **return** 0;
9. }

Output

Address of number variable is fff4

Address of p variable is fff4

Linkedin@KeerthanaGaliveeti

Value of p variable is 50

Pointer to array

1. **int** arr[10];
2. **int** *p[10]=&arr; *// Variable p of type pointer is pointing to the address of an integer array arr.*

Pointer to a function

1. **void** show (**int**);
2. **void**(*p)(**int**) = &show; *// Pointer p is pointing to the address of a function*

Pointer to structure

1. **struct** st {
2.   **int** i;
3.   **float** f;
4. }ref;
5. **struct** st *p = &ref;
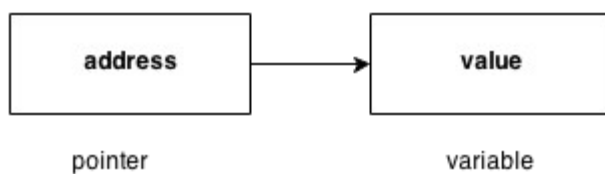
Advantage of pointer

1) Pointer **reduces the code** and **improves the performance**, it is used to retrieving strings, trees, etc. and used with arrays, structures, and functions.

2) We can **return multiple values from a function** using the pointer.

3) It makes you able to **access any memory location** in the computer's memory.

Usage of pointer

There are many applications of pointers in c language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where the pointer is used.

2) Arrays, Functions, and Structures

Pointers in c language are widely used in arrays, functions, and structures. It reduces the code and improves the performance.

Address Of (&) Operator

The address of operator '&' returns the address of a variable. But, we need to use %u to display the address of a variable.

1. #include<stdio.h>
2. **int** main(){
3. **int** number=50;
4. printf("value of number is %d, address of number is %u",number,&number);
5. **return** 0;
6. }

Output

value of number is 50, address of number is fff4

NULL Pointer

A pointer that is not assigned any value but NULL is known as the NULL pointer. If you don't have any address to be specified in the pointer at the time of declaration, you can assign NULL value. It will provide a better approach.

int *p=NULL;

In the most libraries, the value of the pointer is 0 (zero).

Pointer Program to swap two numbers without using the 3rd variable.

```
1.  #include<stdio.h>
2.  int main(){
3.  int a=10,b=20,*p1=&a,*p2=&b;
4.  printf("Before swap: *p1=%d *p2=%d",*p1,*p2);
5.  *p1=*p1+*p2;
6.  *p2=*p1-*p2;
7.  *p1=*p1-*p2;
8.  printf("\nAfter swap: *p1=%d *p2=%d",*p1,*p2);
9.
10. return 0;
11. }
```

Output

Before swap: *p1=10 *p2=20

After swap: *p2=10 *p1=20

Declaration of a function pointer

Till now, we have seen that the functions have addresses, so we can create pointers that can contain these addresses, and hence can point them.

Syntax of function pointer

1. **return** type (*ptr_name)(type1, type2…);

For example:

1. **int** (*ip) (**int**);

In the above declaration, *ip is a pointer that points to a function which returns an int value and accepts an integer value as an argument.

1. **float** (*fp) (**float**);

In the above declaration, ***fp** is a pointer that points to a function that returns a float value and accepts a float value as an argument.

We can observe that the declaration of a function is similar to the declaration of a function pointer except that the pointer is preceded by a **'*'**. So, in the above declaration, fp is declared as a function rather than a pointer.

Till now, we have learnt how to declare the function pointer. Our next step is to assign the address of a function to the function pointer.

1. **float** (*fp) (**int** , **int**);   // Declaration of a function pointer.
2. **float** func(**int** , **int**);   // Declaration of function.
3. fp = &func;   // Assigning address of func to the fp pointer.

In the above declaration, **'fp'** pointer contains the address of the **'func'** function.

Calling a function through a function pointer

We already know how to call a function in the usual way. Now, we will see how to call a function using a function pointer.

Suppose we declare a function as given below:

1. **float** func(**int** , **int**);   // Declaration of a function.

Calling an above function using a usual way is given below:

1. result = func(a , b);   // Calling a function using usual ways.

Calling a function using a function pointer is given below:

1. result = (*fp)(a , b);   // Calling a function using function pointer.

Or

1. result = fp(a , b);   // Calling a function using function pointer, and indirection operator can be removed.

The effect of calling a function by its name or function pointer is the same. If we are using the function pointer, we can omit the indirection operator as we did in the second case. Still, we use the indirection operator as it makes it clear to the user that we are using a function pointer.

Let's understand the function pointer through an example.

1. #include <stdio.h>
2. **int** add(**int**,**int**);
3. **int** main()
4. {
5.   **int** a,b;
6.   **int** (*ip)(**int**,**int**);
7.   **int** result;
8.   printf("Enter the values of a and b : ");

9. scanf("%d %d",&a,&b);

10. ip=add;

11. result=(*ip)(a,b);

12. printf("Value after addition is : %d",result);

13. **return** 0;

14. }

15. **int** add(**int** a,**int** b)

16. {

17. **int** c=a+b;

18. **return** c;

19. }

Output

Passing a function's address as an argument to other function

We can pass the function's address as an argument to other functions in the same way we send other arguments to the function.

Let's understand through an example.

1. include <stdio.h>

2. **void** func1(**void** (*ptr)());

3. **void** func2();

4. **int** main()

5. {
6.   func1(func2);
7.   **return** 0;
8. }
9. **void** func1(**void** (*ptr)())
10. {
11.   printf("Function1 is called");
12.   (*ptr)();
13. }
14. **void** func2()
15. {
16.   printf("\nFunction2 is called");
17. }

In the above code, we have created two functions, i.e., func1() and func2(). The func1() function contains the function pointer as an argument. In the main() method, the func1() method is called in which we pass the address of func2. When func1() function is called, 'ptr' contains the address of 'func2'. Inside the func1() function, we call the func2() function by dereferencing the pointer 'ptr' as it contains the address of func2.

Output


```
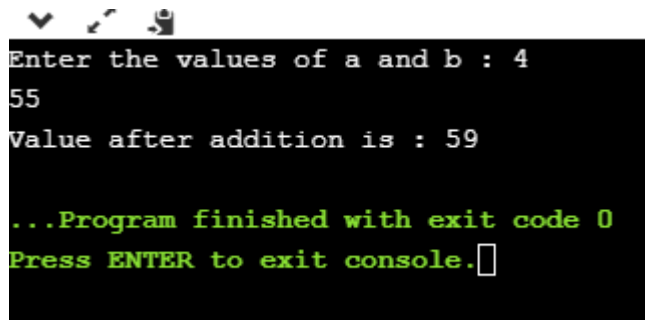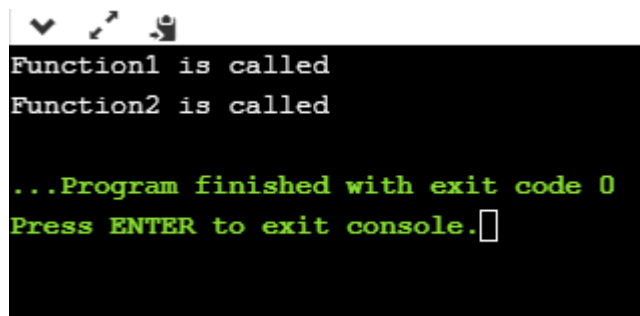Function1 is called
Function2 is called

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

Array of Function Pointers

Function pointers are used in those applications where we do not know in advance which function will be called. In an array of function pointers, array takes the addresses of different functions, and the appropriate function will be called based on the index number.

Let's understand through an example.

1. #include <stdio.h>
2. **float** add(**float,int**);
3. **float** sub(**float,int**);
4. **float** mul(**float,int**);
5. **float** div(**float,int**);
6. **int** main()
7. {
8.   **float** x;   // variable declaration.
9.   **int** y;
10.   **float** (*fp[4]) (**float,int**);   // function pointer declaration.
11.   fp[0]=add;   // assigning addresses to the elements of an array of a function pointer.
12.   fp[1]=sub; {add,sub,mul,div}
13.   fp[2]=mul;
14.   fp[3]=div;
15.   printf("Enter the values of x and y :");
16.   scanf("%f %d",&x,&y);
17. **float** r=(*fp[0]) (x,y);   // Calling add() function.
18.   printf("\nSum of two values is : %f",r);
19.   r=(*fp[1]) (x,y);   // Calling sub() function.
20.   printf("\nDifference of two values is : %f",r);
21.   r=(*fp[2]) (x,y);   // Calling mul() function.
22.   printf("\nMultiplication of two values is : %f",r);
23.   r=(*fp[3]) (x,y);   // Calling div() function.

```
24.     printf("\nDivision of two values is : %f",r);
25.     return 0;
26. }
27.
28. float add(float x,int y)
29. {
30.     float a=x+y;
31.     return a;
32. }
33. float sub(float x,int y)
34. {
35.     float a=x-y;
36.     return a;
37. }
38. float mul(float x,int y)
39. {
40.     float a=x*y;
41.     return a;
42. }
43. float div(float x,int y)
44. {
45.     float a=x/y;
46.     return a;
47. }
```

In the above code, we have created an array of function pointers that contain the addresses of four functions. After storing the addresses of functions in an array of function pointers, we call the functions using the function pointer.

Output

```
Enter the values of x and y :5
7

Sum of two values is : 12.000000
Difference of two values is : -2.000000
Multiplication of two values is : 35.000000
Division of two values is : 0.714286

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

How Are Pointers Related to Arrays

The memory address of the first element is the same as the name of the array:

EXAMPLE PROGRAM :

    int myNumbers[4] = {25, 50, 75, 100};

        // Get the memory address of the myNumbers array

```
printf("%p\n", myNumbers);
```

// Get the memory address of the first array element

```
printf("%p\n", &myNumbers[0]);
```

OUTPUT:

```
0x7ffe70f9d8f0


0x7ffe70f9d8f0
```

```
int myNumbers[4] = {25, 50, 75, 100};
```

// Get the value of the first element in myNumbers

```
printf("%d", *myNumbers);
```

Result:

```
25
```

To access the rest of the elements in myNumbers, you can increment the pointer/array (+1, +2, etc):

```
int myNumbers[4] = {25, 50, 75, 100};
```

// Get the value of the second element in myNumbers

```
printf("%d\n", *(myNumbers + 1));
```

// Get the value of the third element in myNumbers

```c
printf("%d", *(myNumbers + 2));
```

```
50

75
```

```c
int myNumbers[4] = {25, 50, 75, 100};
int *ptr = myNumbers;
int i;
for (i = 0; i < 4; i++) {
  printf("%d\n", *(ptr + i));
}
```

Result:

```
25

50

75

100
```

It is also possible to change the value of array elements with pointers:

```c
int myNumbers[4] = {25, 50, 75, 100};
    // Change the value of the first element to 13
*myNumbers = 13; ----> {13, 17, 34,
    // Change the value of the second element to 17
*(myNumbers +1) = 17;  *(myNumbers + 2) = 34;
    // Get the value of the first element
printf("%d\n", *myNumbers); ---first element
```

```c
// Get the value of the second element

printf("%d\n", *(myNumbers + 1)); second element
```

```
13


17
```

POINTERS & ARRAY OF STRUCTURE

Create an array of structure variable

In the following example we are considering the student structure that we created in the previous tutorial and we are creating an array of student structure variable std of size 3 to hold details of three students.

// student structure

struct student {

char id[15];

  char firstname[64];

  char lastname[64];

  float points;

};

// student structure variable

struct student std[3];

Create pointer variable for structure

Now we will create a pointer variable that will hold the starting address of the student structure variable std.

// student structure pointer variable

struct student *ptr = NULL;

// assign std to ptr

ptr = std;

Accessing each element of the structure array variable via pointer

For this we will first set the pointer variable ptr to point at the starting memory location of std variable. For this we write ptr = std;.

Then, we can increment the pointer variable using increment operator ptr++ to make the pointer point at the next element of the structure array variable i.e., from str[0] to str[1].

We will loop three times as there are three students. So, we will increment pointer variable twice. First increment will move pointer ptr from std[0] to std[1] and the second increment will move pointer ptr from std[1] to std[2].

To reset the pointer variable ptr to point at the starting memory location of structure variable std we write ptr = std;.

Complete code

```
#include <stdio.h>
int main(void) {
  // student structure
  struct student {
  char id[15];
    char firstname[64];
    char  lastname[64];
    float points;
  };

  // student structure variable
  struct student std[3];
```

```c
// student structure pointer variable
struct student *ptr = NULL;


// other variables
int i;
// assign std to ptr
ptr = std;
// get detail for user
for (i = 0; i < 3; i++) {
  printf("Enter detail of student #%d\n", (i + 1));
  printf("Enter ID: ");
  scanf("%s", ptr->id);
  printf("Enter first name: ");
  scanf("%s", ptr->firstname);
  printf("Enter last name: ");
  scanf("%s", ptr->lastname);
  printf("Enter Points: ");
  scanf("%f", &ptr->points);

  // update pointer to point at next element
  // of the array std
  ptr++;
}
// reset pointer back to the starting
// address of std array
ptr = std;

for (i = 0; i < 3; i++) {
```

```c
        printf("\nDetail of student #%d\n", (i + 1));
        // display result via std variable
        printf("\nResult via std\n");
        printf("ID: %s\n", std[i].id);
        printf("First Name: %s\n", std[i].firstname);
        printf("Last Name: %s\n", std[i].lastname);
        printf("Points: %f\n", std[i].points);
        // display result via ptr variable
        printf("\nResult via ptr\n");
        printf("ID: %s\n", ptr->id);
        printf("First Name: %s\n", ptr->firstname);
        printf("Last Name: %s\n", ptr->lastname);
        printf("Points: %f\n", ptr->points);

        // update pointer to point at next element
        // of the array std
        ptr++;
    }
    return 0;
}
```

Output:

Enter detail of student #1

Enter ID: s01

Enter first name: Yusuf

Enter last name: Shakeel

Enter Points: 8

Enter detail of student #2

Enter ID: s02

Enter first name: Jane

Enter last name: Doe

Enter Points: 9

Enter detail of student #3

Enter ID: s03

Enter first name: John

Enter last name: Doe

Enter Points: 6

Detail of student #1

Result via std

ID: s01

First Name: Yusuf

Last Name: Shakeel

Points: 8.000000

Result via ptr

ID: s01

First Name: Yusuf

Last Name: Shakeel

Points: 8.000000

Detail of student #2

Result via std

ID: s02

First Name: Jane

Last Name: Doe

Points: 9.000000

Result via ptr

ID: s02

First Name: Jane

Last Name: Doe

Points: 9.000000

Detail of student #3

Result via std

ID: s03

First Name: John

Last Name: Doe

Points: 6.000000

Result via ptr

ID: s03

First Name: John

Last Name: Doe

Points: 6.000000

C Strings

The string can be defined as the one-dimensional array of characters terminated by a null ('\0').

The character array or the string is used to manipulate text such as word or sentences.

There are two ways to declare a string in c language.

1. By char array

2. By string literal

3. #include<stdio.h>
4. #include <string.h>
5. **int** main(){
6.   **char** ch[11]={'’H’,’o’,’w’,’a’,’r’,’e’,’y’,’o’,’u’, '\0'};
7.   **char** ch2[11]="How are you";
8.   printf("Char Array Value is: %s\n", ch);
9.   printf("String Literal Value is: %s\n", ch2);
10. **return** 0;
11. }

Traversing string is somewhat different from the traversing an integer array. We need to know the length of the array to traverse an integer array, whereas we may use the null character in the case of string to identify the end the string and terminate the loop.

Hence, there are two ways to traverse a string.

- By using the length of string

- By using the null character.

- #include<stdio.h>

- **void** main ()

- {

-   **char** s[] = "hihello";

-   **int** i = 0;

-   **int** count = 0;

-   **while**(i<7)

-   {

- if(s[i]=='a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'u' || s[i] == 'o')

- {

- count ++;

- }

- i++;

- }

- printf("The number of vowels %d",count);

- }

OUTPUT :

The number of vowels 4

Using the null character

Let's see the same example of counting the number of vowels by using the null character.

1. #include<stdio.h>
2. **void** main ()
3. {
4. **char** s[11] = "javatpoint";
5. **int** i = 0;
6. **int** count = 0;
7. **while**(s[i] != NULL)
8. {
9. **if**(s[i]=='a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'u' || s[i] == 'o')
10. {
11. count ++;
12. }
13. i++;
14. }

15.   printf("The number of vowels %d",count);
16. }

Output

The number of vowels 4

Accepting string as the input

Till now, we have used scanf to accept the input from the user. However, it can also be used in the case of strings but with a different scenario. Consider the below code which stores the string while space is encountered.

1. #include<stdio.h>
2. **void** main ()
3. {
4.   **char** s[20];
5.   printf("Enter the string?");
6.   scanf("%s",&s);
7.   printf("You entered %s",s);
8. }

Output

Enter the string?javatpoint is the best

You entered javatpoint

It is clear from the output that, the above code will not work for space separated strings. To make this code working for the space separated strings, the minor changed required in the scanf

function, i.e., instead of writing scanf("%s",s), we must write: scanf("%[^\n]s",s) which instructs the compiler to store the string s while the new line (\n) is encountered. Let's consider the following example to store the space-separated strings.

1. #include<stdio.h>
2. **void** main ()
3. {
4.   **char** s[20];
5.   printf("Enter the string?");
6.   scanf("%[^\n]s",s);
7.   printf("You entered %s",s);
8. }

Output

Enter the string? Hi hello c program

Hi hello c program

Pointers with strings

We have used pointers with the array, functions, and primitive data types so far. However, pointers can be used to point to the strings. There are various advantages of using pointers to point strings. Let us consider the following example to access the string via the pointer.

1. #include<stdio.h>
2. **void** main ()
3. {
4.   **char** s[11] = "Programming";
5.   **char** *p = s; // pointer p is pointing to string s.
6.   printf("%s",p); //
7. }

Output

Programming

As we know that string is an array of characters, the pointers can be used in the same way they were used with arrays. In the above example, p is declared as a pointer to the array of characters s. P affects similar to s since s is the base address of the string and treated as a pointer internally. However, we can not change the content of s or copy the content of s into another string directly. For this purpose, we need to use the pointers to store the strings. In the following example, we have shown the use of pointers to copy the content of a string into another.

1. #include<stdio.h>
2. **void** main ()
3. {
4.   **char** *p = "hello c";
5.   printf("String p: %s\n",p);
6.   **char** *q;
7.   printf("copying the content of p into q...\n");
8.   q = p;
9.   printf("String q: %s\n",q);
10. }

Output

String p: hello c

copying the content of p into q...

String q: hello c

Once a string is defined, it cannot be reassigned to another set of characters. However, using pointers, we can assign the set of characters to the string. Consider the following example.

1. #include<stdio.h>
2. **void** main ()
3. {
4.   **char** *p = "hello c";
5.   printf("Before assigning: %s\n",p);
6.   p = "hello";
7.   printf("After assigning: %s\n",p);
8. }

Output

Before assigning: hello c

After assigning : hello

C gets() function

The gets() function enables the user to enter some characters followed by the enter key. All the characters entered by the user get stored in a character array. The null character is added to the array to make it a string. The gets() allows the user to enter the space-separated strings. It returns the string entered by the user.

Declaration

1. **char**[] gets(**char**[]);

Reading string using gets()

1. #include<stdio.h>
2. **void** main ()
3. {
4.   **char** s[30];
5.   printf("Enter the string? ");
6.   gets(s);
7.   printf("You entered %s",s);
8. }

Output

Enter the string?

javatpoint is the best

You entered javatpoint is the best

The gets() function is risky to use since it doesn't perform any array bound checking and keep reading the characters until the new line (enter) is encountered. It suffers from buffer overflow, which can be avoided by using fgets(). The fgets() makes sure that not more than the maximum limit of characters are read. Consider the following example.

1. #include<stdio.h>
2. **void** main()
3. {

4. **char** str[20];
5. printf("Enter the string? ");
6. fgets(str, 20, stdin);
7. printf("%s", str);
8. }

Output

Enter the string? javatpoint is the best website

javatpoint is the b

C puts() function

The puts() function is very much similar to printf() function. The puts() function is used to print the string on the console which is previously read by using gets() or scanf() function. The puts() function returns an integer value representing the number of characters being printed on the console. Since, it prints an additional newline character with the string, which moves the cursor to the new line on the console, the integer value returned by puts() will always be equal to the number of characters present in the string plus 1.

Declaration

1. **int** puts(**char**[])

Let's see an example to read a string using gets() and print it on the console using puts().

1. #include<stdio.h>
2. #include <string.h>
3. **int** main(){
4. **char** name[50];

5. printf("Enter your name: ");

6. gets(name); //reads string from user   fgets(name,50,stdin);

7. printf("Your name is: ");

8. puts(name); //displays string

9. **return** 0;

10. }

Output:

Enter your name: Sonoo Jaiswal

Your name is: Sonoo Jaiswal

C String Functions

here are many important string functions defined in "string.h" library.

No.	Function	Description
1)	strlen(string_name)	returns the length of string name.
2)	strcpy(destination,	copies the contents of source string to destination string.

	source)	
3)	strcat(first_string, second_string)	concats or joins first string with second string. The result of the string is stored in first string.
4)	strcmp(first_string, second_string)	compares the first string with the second string. If both strings are the same, it returns 0.
5)	strrev(string)	returns reverse string.
6)	strlwr(string)	returns string characters in lowercase.
7)	strupr(string)	returns string characters in uppercase

1. #include<stdio.h>
2. #include <string.h>
3. **int** main(){
4. **char** ch[20]={'c', 'a', 'v', 'a', 't', 'i', 'o', 'i', 'u', 't', '\0'};
5.   printf("Length of string is: %d",strlen(ch));
6.   **return** 0;
7. }

Output:

Length of string is: 10

1. #include<stdio.h>
2. #include <string.h>
3. **int** main(){
4.   **char** ch[20]={'c', 'a', 'v', 'a', 't', 'p', 'o', 'm', 'n', 't', '\0'};
5.   **char** ch2[20];
6.   strcpy(ch2,ch);
7.   printf("Value of second string is: %s",ch2);
8.   **return** 0;
9.   }

Output:

Value of second string is: cavatpomnt

1. #include<stdio.h>
2. #include <string.h>
3. **int** main(){
4.   **char** ch[10]={'h', 'e', 'l', 'l', 'o', '\0'};
5.   **char** ch2[10]={'c', '\0'};
6.   strcat(ch,ch2);
7.   printf("Value of first string is: %s",ch);
8.   **return** 0;
9.   }

Output

Value of first string is: helloc

1. #include<stdio.h>
2. #include <string.h>

3. **int** main(){

4.   **char** str1[20],str2[20];

5.   printf("Enter 1st string: ");

6.   gets(str1);//reads string from console

7.   printf("Enter 2nd string: ");

8.   gets(str2);

9.   **if**(strcmp(str1,str2)==0)

10.   printf("Strings are equal");

11. **else**

12.   printf("Strings are not equal");

13. **return** 0;

14. }

Output

Enter 1st string: hello

Enter 2nd string: hello

Strings are equal

1. #include<stdio.h>

2. #include <string.h>

3. **int** main(){

4.   **char** str[20];

5.   printf("Enter string: ");

6.   gets(str);//reads string from console

7.   printf("String is: %s",str);

8.   printf("\nReverse String is: %s",strrev(str));

9.   **return** 0;

10. }

Syntax:

It has the following syntax:

1. char* strlwr(char* str);
2. #include<stdio.h>
3. #include <string.h>
4. int main(){
5.   char str[20];
6.   printf("Enter string: ");
7.   gets(str);//reads string from console
8.   printf("String is: %s",str);
9.   printf("\nLower String is: %s",strlwr(str));   HFGJHN- hfgjhn
10. return 0;
11. }

12. #include<stdio.h>
13. #include <string.h>
14. int main(){
15.   char str[20];
16.   printf("Enter string: ");
17.   gets(str);//reads string from console
18.   printf("String is: %s",str);
19.   printf("\nUpper String is: %s",strupr(str));
20.   return 0;
21. }

Syntax:

It has the following syntax:

1. char *strstr(const char *string, const char *match);

Here is an explanation of each heading or aspect related to the *strstr() function*:

Function Name:

The *strstr() function* stands for *"string search"*. It is a built-in C library function that searches for a substring within a *larger string*.

Return Type:

The function returns a pointer to the substring *('match')*'s first occurrence within the text *('string')*. If the substring cannot be found, the method returns *NULL*.

Parameters:

- ○ 'string' (const char *): It is the input string in which the substring should be found.
- ○ 'match' (const char *): It is the substring you want to search for within the string.

Function Behavior:

The *strstr() function* looks for the first instance of the substring *('match')* inside the more extensive string *('string')*.

It returns a reference to the *haystack place* where the substring begins or *NULL* if it cannot be found.

Case Sensitivity:

By default, *strstr()* is case-sensitive. It differentiates between *uppercase* and *lowercase* characters in both the string and the match.

Example Usage:

1. #include <stdio.h>

2. #include <string.h>

3. intmain() {

4. char str[100] = "this is meena with c and java";

5. char *sub;

6.

7. sub = strstr(str, "java");

8.

9. if (sub != NULL) {

10. printf("Substring is: %s", sub);

11. } else {

12. printf("Substring not found.");

13. }

14.

15. return 0;

16. }

Output:

Substring is:meena with c and java

Command Line Arguments in C

The arguments passed from command line are called command line arguments. These arguments are handled by main() function.

To support command line argument, you need to change the structure of main() function as given below.

1. **int** main(**int** argc, **char** *argv[])

Here, **argc** counts the number of arguments. It counts the file name as the first argument.

The **argv[]** contains the total number of arguments. The first argument is the file name always.

Example

Let's see the example of command line arguments where we are passing one argument with file name.

```
 1.  #include <stdio.h>  – simple.c                           simple.c welcome program 3
 2.   void main(int argc, char *argv[] )  {
 3.    printf("Program name is: %s\n", argv[0]);
 4.    if(argc < 2){
 5.      printf("No argument passed through command line.\n");
 6.    }
 7.    else{
 8.      printf("First argument is: %s\n", argv[1]);
 9.    }
10. }
```

Run this program as follows in Linux:

```
 1.  ./program hello
```

Run this program as follows in Windows from command line:

```
 1.  program.exe hello
```

Output:

Program name is: program

First argument is: hello

If you pass many arguments, it will print only one.

1. ./program hello c how r u

Output:

Program name is: program

First argument is: hello

But if you pass many arguments within double quote, all arguments will be treated as a single argument only.

1. ./program "hello c how r u"

Output:

Program name is: program

First argument is: hello c how r u

C Preprocessor Directives

The C preprocessor is a micro processor that is used by compiler to transform your code before compilation. It is called micro preprocessor because it allows us to add macros.

Linkedin@KeerthanaGaliveeti
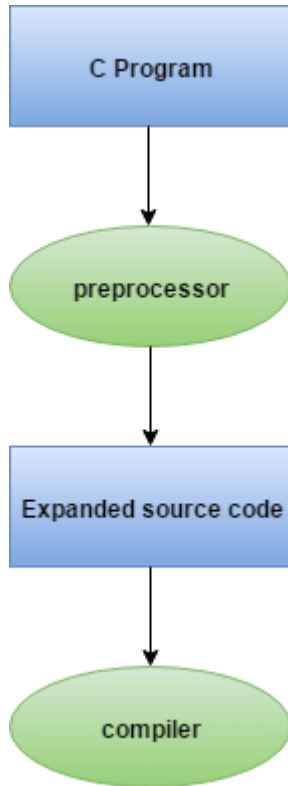
All preprocessor directives starts with hash # symbol.

Let's see a list of preprocessor directives.

- ○ #include

- ○ #define

- ○ #undef

- ○ #ifdef

- ○ #ifndef

- ○ #if

- ○ #else

- ○ #elif

- ○ #endif

- ○ #error

○ #pragma

C Macros

C macros provide a potent method for code reuse and simplification. They let programmers construct **symbolic names** or phrases that are changed to certain values before the compilation process begins. The use of more **macros** makes code easier to **comprehend, maintain**, and makes mistakes less likely. In this article, we'll delve deeper into the concept of C macros and cover their advantages, ideal usage scenarios, and potential hazards.

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

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

Object-like Macros

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

1. #define PI 3.14

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

Function-like Macros

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

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

Here, MIN is the macro name.

Visit #define to see the full example of object-like and function-like macros.

C Predefined Macros

ANSI C defines many predefined macros that can be used in c program.

No.	Macro	Description
1	_DATE_	represents current date in "MMM DD YYYY" format.
2	_TIME_	represents current time in "HH:MM:SS" format.
3	_FILE_	represents current file name.
4	_LINE_	represents current line number.
5	_STDC_	It is defined as 1 when compiler complies with the ANSI standard.

C predefined macros example

File: simple.c

1. #include<stdio.h>
2. **int** main(){

3. printf("File :%s\n", __FILE__);
4. printf("Date :%s\n", __DATE__);
5. printf("Time :%s\n", __TIME__);
6. printf("Line :%d\n", __LINE__);
7. printf("STDC :%d\n", __STDC__);
8. **return** 0;
9. }

Output:

File :simple.c

Date :Dec 6 2015

Time :12:28:46

Line :6

STDC :1

Advantages of Using Macros:

There are various advantages of Macros in C. Some main advantages of C macros are as follows:

Code reuse: By allowing developers to declare a piece of code just once and use it several times, *macros* help to promote modular programming and minimize code duplication.

Code abbreviation: *Macros* make it possible to *write clear, expressive code* that is simpler to read and comprehend the intentions of the programmer.

Performance Optimization: By minimizing *function call overhead*, macros may be utilized to *optimize code execution*. For instance, it is possible to inline brief pieces of code using function-like macros.

Using **macros, conditional compilation** enables distinct sections of the code to be **included** or **removed** based on **predetermined circumstances. Debugging** or **platform-specific code** both benefit from this functionality.

When Using Macros, Exercise Caution:

Use **caution** while constructing function-like **macros** in brackets. Always use brackets to contain **parameters** and the full **macro body** to avoid unexpected outcomes brought on by operator precedence.

Macro Side consequences: *Steer clear* of macros with **negative consequences. Multiple evaluations** of macro arguments may result in surprising results since macros are directly substituted.

Use capital letters to distinguish macro names from standard C identifiers and to make the code easier to understand.

Macros' best practices:

There are various practices of C macros. Some of them are as follows:

For object-like macros, use constants: If possible, use *object-like macros* for constants to make the code more *readable* and make any necessary adjustments simpler.

Macros with Functions for Basic Operations: Use *macros* that resemble functions for straightforward, *one-line operations*. Use normal functions in their place for more intricate processes.

Use Predefined Macros Wisely: While predefined macros like __*DATE__, __TIME__, and __FILE__*are useful for debugging and logging, don't rely on them for essential functions.

Macros and Conditional Compilation:

When **conditional compilation** is used, macros significantly contribute utilizing the ***#ifdef, #ifndef, #else, and #endif*** directives. It gives the compiler the ability to include or reject code blocks depending on precise criteria.

Using Inline Functions vs. Macros:

Inline functions offer comparable advantages to macros while avoiding some of the possible drawbacks of macros. Macros can enhance speed since they directly replace code. Modern compilers may frequently inline appropriate functions, making them a safer option in many circumstances.

Using macros for debugging:

As shown in the example above, macros can be helpful in debugging by supplying extra details like ***file names, line numbers***, and ***timestamps***.

C #include

The #include preprocessor directive is used to paste code of given file into current file. It is used include system-defined and user-defined header files. If included file is not found, compiler renders error.

By the use of #include directive, we provide information to the preprocessor where to look for the header files. There are two variants to use #include directive.

1. #include <filename>
2. #include "filename"

The **#include <filename>** tells the compiler to look for the directory where system header files are held. In UNIX, it is \usr\include directory.

The **#include "filename"** tells the compiler to look in the current directory from where program is running.

#include directive example

Let's see a simple example of #include directive. In this program, we are including stdio.h file because printf() function is defined in this file.

1. #include<stdio.h>
2. **int** main(){
3.   printf("Hello C");
4.   **return** 0;
5. }

Output:

Hello C

#include notes:

Note 1: In #include directive, comments are not recognized. So in case of #include <a//b>, a//b is treated as filename.

Note 2: In #include directive, backslash is considered as normal text not escape sequence. So in case of #include <a\nb>, a\nb is treated as filename.

Note 3: You can use only comment after filename otherwise it will give error.

C #define

The C programming language's ***#define preprocessor directive*** provides a strong and flexible tool for declaring ***constants*** and producing ***macros***. It conducts textual replacement before actual compilation during the pre-processing phase of C programs, where it plays a significant role. By using this functionality, developers may improve their code's ***readability, maintainability***, and ***effectiveness***.

Developers may give meaningful names to ***fixed values*** when declaring constants using ***#define***, which makes the code easier to understand and maintain. Programmers can avoid hard coding values throughout the code by using ***constants***, which reduces errors and ensures ***consistency***.

Additionally, ***#define*** makes it possible to create ***macros***, which serve as ***code blocks***. As they substitute ***function calls*** and sometimes provide more control over program behavior, macros aid in constructing short, effective lines of code. However, ***macros*** must be used carefully since they are directly substituted in the code and, if incorrectly specified, might have unexpected outcomes.

The #define preprocessor directive is used to define constant or micro substitution. It can use any basic data type.

Syntax:

1. #define token value

Let's see an example of #define to define a constant.

1. #include <stdio.h>
2. #define PI 3.14
3. main() {
4.   printf("%f",PI);
5. }

Output:

3.140000

Explanation:

In this example, we define a constant **PI** with a value of **3.14**. After that, the **printf() function** uses the **PI constant** to display the value. This program's **output** after compilation and execution is as follows:

Let's see an example of #define to create a macro.

1. #include <stdio.h>
2. #define MIN(a,b) ((a)<(b)?(a):(b))
3. **void** main() {
4.   printf("Minimum between 10 and 20 is: %d\n", MIN(10,20));
5. }

Output:

Minimum between 10 and 20 is: 10

Explanation:

In this example, we develop a **MIN macro** that accepts the two inputs **a** and **b**. The macro's definition returns the lowest value between the two inputs. The preprocessor replaces the **MIN macro** with the actual code implementation when it is used with the **inputs (10, 20),** resulting in **((10) (20)? (10): (20))**. It is equal to 10, which shows in the output.

C #undef

The #undef preprocessor directive is used to undefine the constant or macro defined by #define.

Syntax:

1. #undef token

Let's see a simple example to define and undefine a constant.

1. #include <stdio.h>
2. #define PI 3.14
3. #undef PI
4. main() {
5.   printf("%f",PI);
6. }

Output:

Compile Time Error: 'PI' undeclared

The #undef directive is used to define the preprocessor constant to a limited scope so that you can declare constant again.

Let's see an example where we are defining and undefining number variable. But before being undefined, it was used by square variable.

1. #include <stdio.h>
2. #define number 15
3. **int** square=number*number;
4. #undef number
5. main() {
6.   printf("%d",square);
7. }

Output:

225

C #ifdef

The #ifdef preprocessor directive checks if macro is defined by #define. If yes, it executes the code otherwise #else code is executed, if present.

Syntax:

1. #ifdef MACRO
2. //code
3. #endif

Syntax with #else:

1. #ifdef MACRO
2. //successful code
3. #else
4. //else code
5. #endif

C #ifdef example

Let's see a simple example to use #ifdef preprocessor directive.

1. #include <stdio.h>
2. #include <conio.h>
3. #define NOINPUT
4. **void** main() {
5. **int** a=0;
6. #ifdef NOINPUT
7. a=2;
8. #else

9. printf("Enter a:");

10. scanf("%d", &a);

11. #endif

12. printf("Value of a: %d\n", a);

13. getch();

14. }

Output:

Value of a: 2

But, if you don't define NOINPUT, it will ask user to enter a number.

1. #include <stdio.h>

2. #include <conio.h>

3. **void** main() {

4. **int** a=0;

5. #ifdef NOINPUT

6. a=2;

7. #else

8. printf("Enter a:");

9. scanf("%d", &a);

10. #endif

11. printf("Value of a: %d\n", a);

12. getch();

13. }

Output:

Enter a:5

Value of a: 5

C #ifndef

The #ifndef preprocessor directive checks if macro is not defined by #define. If yes, it executes the code otherwise #else code is executed, if present.

Syntax:

1. #ifndef MACRO
2. //code
3. #endif

Syntax with #else:

1. #ifndef MACRO
2. //successful code
3. #else
4. //else code
5. #endif

C #ifndef example

Let's see a simple example to use #ifndef preprocessor directive.

1. #include <stdio.h>
2. #include <conio.h>
3. #define INPUT
4. **void** main() {
5. **int** a=0;
6. #ifndef INPUT
7. a=2;
8. #else

9. printf("Enter a:");

10. scanf("%d", &a);

11. #endif

12. printf("Value of a: %d\n", a);

13. getch();

14. }

Output:

Enter a:5

Value of a: 5

1. #include <stdio.h>

2. #include <conio.h>

3. **void** main() {

4. **int** a=0;

5. #ifndef INPUT

6. a=2;

7. #else

8. printf("Enter a:");

9. scanf("%d", &a);

10. #endif

11. printf("Value of a: %d\n", a);

12. getch();

13. }

Output:

Value of a: 2

C #if

The #if preprocessor directive evaluates the expression or condition. If condition is true, it executes the code otherwise #elseif or #else or #endif code is executed.

Syntax:

1. #if expression
2. //code
3. #endif

Syntax with #else:

1. #if expression
2. //if code
3. #else
4. //else code
5. #endif

Syntax with #elif and #else:

1. #if expression
2. //if code
3. #elif expression
4. //elif code
5. #else
6. //else code
7. #endif

C #if example

Let's see a simple example to use #if preprocessor directive.

1. #include <stdio.h>

2. #include <conio.h>

3. #define NUMBER 0

4. **void** main() {

5. #if (NUMBER==0)

6. printf("Value of Number is: %d",NUMBER);

7. #endif

8. getch();

9. }

Output:

Value of Number is: 0

Let's see another example to understand the #if directive clearly.

1. #include <stdio.h>

2. #include <conio.h>

3. #define NUMBER 1

4. **void** main() {

5. clrscr();

6. #if (NUMBER==0)

7. printf("1 Value of Number is: %d",NUMBER);

8. #endif

9.

10. #if (NUMBER==1)

11. printf("2 Value of Number is: %d",NUMBER);

12. #endif

13. getch();

14. }

Output:

C #if

The #if preprocessor directive evaluates the expression or condition. If condition is true, it executes the code otherwise #elseif or #else or #endif code is executed.

Syntax:

1. #if expression
2. //code
3. #endif

Syntax with #else:

1. #if expression
2. //if code
3. #else
4. //else code
5. #endif

Syntax with #elif and #else:

1. #if expression
2. //if code
3. #elif expression
4. //elif code
5. #else
6. //else code
7. #endif

C #if example

Let's see a simple example to use #if preprocessor directive.

1. #include <stdio.h>
2. #include <conio.h>
3. #define NUMBER 0
4. **void** main() {
5. #if (NUMBER==0)
6. printf("Value of Number is: %d",NUMBER);
7. #endif
8. getch();
9. }

Output:

Value of Number is: 0

Let's see another example to understand the #if directive clearly.

1. #include <stdio.h>
2. #include <conio.h>
3. #define NUMBER 1
4. **void** main() {
5. clrscr();
6. #if (NUMBER==0)
7. printf("1 Value of Number is: %d",NUMBER);
8. #endif
9.
10. #if (NUMBER==1)
11. printf("2 Value of Number is: %d",NUMBER);
12. #endif
13. getch();

14. }

Output:

2 Value of Number is: 1

C #else

The #else preprocessor directive evaluates the expression or condition if condition of #if is false. It can be used with #if, #elif, #ifdef and #ifndef directives.

Syntax:

1. #if expression
2. //if code
3. #else
4. //else code
5. #endif

Syntax with #elif:

1. #if expression
2. //if code
3. #elif expression
4. //elif code
5. #else
6. //else code
7. #endif

C #else example

Let's see a simple example to use #else preprocessor directive.

1. #include <stdio.h>

2. #include <conio.h>

3. #define NUMBER 1

4. **void** main() {

5. #if NUMBER==0

6. printf("Value of Number is: %d",NUMBER);

7. #else

8. print("Value of Number is non-zero");

9. #endif

10. getch();

11. }

Output:

Value of Number is non-zero

C #error

The #error preprocessor directive indicates error. The compiler gives fatal error if #error directive is found and skips further compilation process.

C #error example

Let's see a simple example to use #error preprocessor directive.

1. #include<stdio.h>

2. #ifndef __MATH_H

3. #error First include then compile

4. #else

5. **void** main(){

6.   **float** a;

7.   a=sqrt(7);

8.   printf("%f",a);

9. }

10.#endif

Output:

Compile Time Error: First include then compile

But, if you include math.h, it does not gives error.

1. #include<stdio.h>
2. #include<math.h>
3. #ifndef __MATH_H
4. #error First include then compile
5. #else
6. **void** main(){
7.   **float** a;
8.   a=sqrt(7);
9.   printf("%f",a);
10. }
11. #endif

Output:

2.645751

C #pragma

The #pragma preprocessor directive is used to provide additional information to the compiler. The #pragma directive is used by the compiler to offer machine or operating-system feature.

Syntax:

1. #pragma token

Different compilers can provide different usage of #pragma directive.

The turbo C++ compiler supports following #pragma directives.

1. #pragma argsused
2. #pragma exit
3. #pragma hdrfile
4. #pragma hdrstop
5. #pragma inline
6. #pragma option
7. #pragma saveregs
8. #pragma startup
9. #pragma warn

Let's see a simple example to use #pragma preprocessor directive.

1. #include<stdio.h>
2. #include<conio.h>
3.
4. **void** func() ;
5.
6. #pragma startup func
7. #pragma exit func
8.
9. **void** main(){
10. printf("\nI am in main");
11. getch();
12. }

13.

14. **void** func(){

15. printf("\nI am in func");

16. getch();

17. }

Output:

I am in func

I am in main

I am in func

File Handling in C

In programming, we may require some specific input data to be generated several numbers of times. Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console, and since the memory is volatile, it is impossible to recover the programmatically generated data again and again. However, if we need to do so, we may store it onto the local file system which is volatile and can be accessed every time. Here, comes the need of file handling in C.

File handling in C enables us to create, update, read, and delete the files stored on the local file system through our C program. The following operations can be performed on a file.

- ○ Creation of the new file

- ○ Opening an existing file
- ○ Reading from the file
- ○ Writing to the file
- ○ Deleting the file

Functions for file handling

There are many functions in the C library to open, read, write, search and close the file. A list of file functions are given below:

No.	Function	Description
1	fopen()	opens new or existing file
2	fprintf()	write data into the file
3	fscanf()	reads data from the file
4	fputc()	writes a character into the file
5	fgetc()	reads a character from file

6	fclose()	closes the file
7	fseek()	sets the file pointer to given position
8	fputw()	writes an integer to file
9	fgetw()	reads an integer from file
10	ftell()	returns current position
11	rewind()	sets the file pointer to the beginning of the file

Opening File: fopen()

We must open a file before it can be read, write, or update. The fopen() function is used to open a file. The syntax of the fopen() is given below.

1. **FILE** *fopen(**const char** * filename, **const char** * mode);

The fopen() function accepts two parameters:

○ The file name (string). If the file is stored at some specific location, then we must mention the path at which the file is stored. For example, a file name can be like **"c://some_folder/some_file.txt"**.

○ The mode in which the file is to be opened. It is a string.

We can use one of the following modes in the fopen() function.

Mode	Description
r	opens a text file in read mode
w	opens a text file in write mode
a	opens a text file in append mode
r+	opens a text file in read and write mode
w+	opens a text file in read and write mode
a+	opens a text file in read and write mode
rb	opens a binary file in read mode

wb	opens a binary file in write mode
ab	opens a binary file in append mode
rb+	opens a binary file in read and write mode
wb+	opens a binary file in read and write mode
ab+	opens a binary file in read and write mode

The fopen function works in the following way.

- ○ Firstly, It searches the file to be opened.
- ○ Then, it loads the file from the disk and place it into the buffer. The buffer is used to provide efficiency for the read operations.
- ○ It sets up a character pointer which points to the first character of the file.

Consider the following example which opens a file in write mode.

1. #include<stdio.h>

```
2.  void main( )
3.  {
4.  FILE *fp ;
5.  char ch ;
6.  fp = fopen("file_handle.c","r") ;
7.  while ( 1 )
8.  {
9.  ch = fgetc ( fp ) ;
10. if ( ch == EOF )
11. break ;
12. printf("%c",ch) ;
13. }
14. fclose (fp ) ;
15. }
```

Output

The content of the file will be printed.

#include;

void main( )

{

FILE *fp; // file pointer

char ch;

fp = fopen("file_handle.c","r");

while (1)

{

ch = fgetc (fp); //Each character of the file is read and stored in the character file.

if (ch == EOF)

break;

printf("%c",ch);

}

fclose (fp);

}

Closing File: fclose()

The fclose() function is used to close a file. The file must be closed after performing all the operations on it. The syntax of fclose() function is given below:

1. **int** fclose(**FILE** *fp);

C fprintf() and fscanf()

Writing File : fprintf() function

The fprintf() function is used to write set of characters into file. It sends formatted output to a stream.

Syntax:

1. **int** fprintf(**FILE** *stream, **const char** *format [, argument, ...])

Example:

1. #include <stdio.h>
2. main(){
3.   **FILE** *fp;
4.   fp = fopen("file.txt", "w");//opening file
5.   fprintf(fp, "Hello file by fprintf...\n");//writing data into file
6.   fclose(fp);//closing file
7. }

Reading File : fscanf() function

The fscanf() function is used to read set of characters from file. It reads a word from the file and returns EOF at the end of file.

Syntax:

1. **int** fscanf(**FILE** *stream, **const char** *format [, argument, ...])

Example:

1. #include <stdio.h>
2. main(){
3.   **FILE** *fp;
4.   **char** buff[255];//creating char array to store data of file
5.   fp = fopen("file.txt", "r");
6.   **while**(fscanf(fp, "%s", buff)!=EOF){
7.   printf("%s ", buff);
8.   }

9. fclose(fp);

10. }

Output:

Hello file by fprintf...

C fputc() and fgetc()

Writing File : fputc() function

The fputc() function is used to write a single character into file. It outputs a character to a stream.

Syntax:

1. **int** fputc(**int** c, **FILE** *stream)

Example:

1. #include <stdio.h>
2. main(){
3.   **FILE** *fp;
4.   fp = fopen("file1.txt", "w");//opening file
5.   fputc('a',fp);//writing single character into file
6.   fclose(fp);//closing file
7. }

file1.txt

a

Reading File : fgetc() function

The fgetc() function returns a single character from the file. It gets a character from the stream. It returns EOF at the end of file.

Syntax:

1. **int** fgetc(**FILE** *stream)

Example:

1. #include<stdio.h>
2. #include<conio.h>
3. **void** main(){
4. **FILE** *fp;
5. **char** c;
6. clrscr();
7. fp=fopen("myfile.txt","r");
8.
9. **while**((c=fgetc(fp))!=EOF){
10. printf("%c",c);
11. }
12. fclose(fp);
13. getch();
14. }

myfile.txt

this is simple text message

C fputs() and fgets()

The fputs() and fgets() in C programming are used to write and read string from stream. Let's see examples of writing and reading file using fgets() and fgets() functions.

Writing File : fputs() function

The fputs() function writes a line of characters into file. It outputs string to a stream.

Syntax:

1. **int** fputs(**const char** *s, **FILE** *stream)

Example:

1. #include<stdio.h>
2. #include<conio.h>
3. **void** main(){
4. **FILE** *fp;
5. clrscr();
6.
7. fp=fopen("myfile2.txt","w");
8. fputs("hello c programming",fp);
9.
10. fclose(fp);
11. getch();
12. }

myfile2.txt

hello c programming

Reading File : fgets() function

The fgets() function reads a line of characters from file. It gets string from a stream.

Syntax:

1. **char*** fgets(**char** *s, **int** n, **FILE** *stream)

Example:

1. #include<stdio.h>
2. #include<conio.h>
3. **void** main(){
4. **FILE** *fp;
5. **char** text[300];
6. clrscr();
7.
8. fp=fopen("myfile2.txt","r");
9. printf("%s",fgets(text,200,fp));
10.
11. fclose(fp);
12. getch();
13. }

Output:

hello c programming

C fseek() function

The fseek() function is used to set the file pointer to the specified offset. It is used to write data into file at desired location.

Syntax:

1. **int** fseek(**FILE** *stream, **long int** offset, **int** whence)

There are 3 constants used in the fseek() function for whence: SEEK_SET, SEEK_CUR and SEEK_END.

Example:

1. #include <stdio.h>

2. **void** main(){

3.   **FILE** *fp;

4.

5.   fp = fopen("myfile.txt","w+");

6.   fputs("This is javatpoint", fp);

7.

8.   fseek(fp, 7, SEEK_SET);

9.   fputs("sonoo jaiswal", fp);

10.   fclose(fp);

11. }

myfile.txt

This is sonoo jaiswal

1. #include <stdio.h>

2.

3. **int** main() {

4. **FILE** *file = fopen("data.txt", "r");

5. **if** (file == NULL) {

6. printf("Unable to open the file.\n");

7. **return** 1;

8. }

9.

10. **char** line[100];

11. fseek(file, 4, SEEK_SET); // Move to the beginning of the fourth line

12. fgets(line, **sizeof**(line), file);

13. printf("Fourth line: %s", line);

14.

15. fclose(file);

16. **return** 0;

17. }

Output:

Fourth line: Line 4

C rewind() function

The rewind() function sets the file pointer at the beginning of the stream. It is useful if you have to use stream many times.

Syntax:

1. **void** rewind(**FILE** *stream)

Example:

File: file.txt

1. this is a simple text

File: rewind.c

1. #include<stdio.h>

2. #include<conio.h>

3. **void** main(){

4. **FILE** *fp;

5. **char** c;

6. clrscr();

7. fp=fopen("file.txt","r");

8.

9. **while**((c=fgetc(fp))!=EOF){

10. printf("%c",c);

11. }

12.

13. rewind(fp);//moves the file pointer at beginning of the file

14.

15. **while**((c=fgetc(fp))!=EOF){

16. printf("%c",c);

17. }

18.

19. fclose(fp);

20. getch();

21. }

Output:

this is a simple textthis is a simple text

C ftell() function

The ftell() function returns the current file position of the specified stream. We can use ftell() function to get the total size of a file after moving file pointer at the end of file. We can use SEEK_END constant to move the file pointer at the end of file.

Syntax:

1. **long int** ftell(**FILE** *stream)

Example:

File: ftell.c

1. #include <stdio.h>

2. #include <conio.h>

3. **void** main (){

4.   **FILE** *fp;

5.   **int** length;

6.   clrscr();

7.   fp = fopen("file.txt", "r");

8.   fseek(fp, 0, SEEK_END);

9.

10.   length = ftell(fp);

11.

12.   fclose(fp);

13.   printf("Size of file: %d bytes", length);

14.   getch();

15. }

Output:

Size of file: 21 bytes

Dynamic memory allocation in C

The concept of **dynamic memory allocation in c language** *enables the C programmer to allocate memory at runtime*. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

1. malloc()

2. calloc()

3. realloc()

4. free()

Before learning above functions, let's understand the difference between static memory allocation and dynamic memory allocation.

static memory allocation	dynamic memory allocation
memory is allocated at compile time.	memory is allocated at run time.
memory can't be increased while executing program.	memory can be increased while executing program.
used in array.	used in linked list.

Now let's have a quick look at the methods used for dynamic memory allocation.

malloc()	allocates single block of requested memory.
calloc()	allocates multiple block of requested memory.
realloc()	reallocates the memory occupied by malloc() or calloc() functions.
free()	frees the dynamically allocated memory.

malloc() function in C

The malloc() function allocates single block of requested memory.

It doesn't initialize memory at execution time, so it has garbage value initially.

It returns NULL if memory is not sufficient.

The syntax of malloc() function is given below:

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

Let's see the example of malloc() function.

1. #include<stdio.h>
2. #include<stdlib.h>
3. **int** main(){
4. **int** n,i,*ptr,sum=0;
5.   printf("Enter number of elements: ");
6.   scanf("%d",&n);
7.   ptr=(**int***)malloc(n***sizeof**(**int**)); //memory allocated using malloc
8.   **if**(ptr==NULL)
9.   {
10.   printf("Sorry! unable to allocate memory");
11.   exit(0);
12. }
13. printf("Enter elements of array: ");
14. **for**(i=0;i<n;++i)
15. {
16.   scanf("%d",ptr+i);

17.   sum+=*(ptr+i);

18. }

19. printf("Sum=%d",sum);

20. free(ptr);

21. **return** 0;

22. }

Output

Enter elements of array: 3

Enter elements of array: 10

10

10

Sum=30
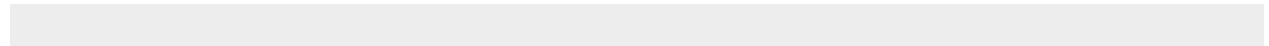
calloc() function in C

The calloc() function allocates multiple block of requested memory.

It initially initialize all bytes to zero.

It returns NULL if memory is not sufficient.

The syntax of calloc() function is given below:

1. ptr=(cast-type*)calloc(number, byte-size)

Let's see the example of calloc() function.

1. #include<stdio.h>

2. #include<stdlib.h>

3. **int** main(){

4.   **int** n,i,*ptr,sum=0;

5.   printf("Enter number of elements: ");

6.   scanf("%d",&n);

7.   ptr=(**int***)calloc(n,**sizeof**(**int**)); //memory allocated using calloc

8.   **if**(ptr==NULL)

9.   {

10.   printf("Sorry! unable to allocate memory");

11.   exit(0);

12.   }

13.   printf("Enter elements of array: ");

14.   **for**(i=0;i<n;++i)

15.   {

16.   scanf("%d",ptr+i);

17.   sum+=*(ptr+i);

18.   }

19.   printf("Sum=%d",sum);

20.   free(ptr);

21. **return** 0;

22. }

Output

Enter elements of array: 3

Enter elements of array: 10

10

10

Sum=30

realloc() function in C

If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.

Let's see the syntax of realloc() function.

1. ptr=realloc(ptr, **new**-size)

free() function in C

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

Let's see the syntax of free() function.

    free(ptr)

ENUM IN C :

The enum in C is also known as the enumerated type. It is a user-defined data type that consists of integer values, and it provides meaningful names to these values. The use of enum in C makes the program easy to understand and maintain. The enum is defined by using the enum keyword.

The following is the way to define the enum in C:

enum flag{integer_const1, integer_const2,. ... integter_constN};

```c
#include <stdio.h>
 enum weekdays{Sunday=1, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
 int main()
{
enum weekdays w; // variable declaration of weekdays type
w=Monday; // assigning value of Monday to w.
printf("The value of w is %d",w);
   return 0;
}
```