**INTRODUCTION**

* The word computer is derived from word compute which means to calculate.
* The process of converting the required input into output form with the help of computer is called as data processing.

**How does a computer work?**

User

Application

Operating System

Hardware

**PROGRAM DEVELOPMENT CYCLE**

* Problem definition
* Problem analysis
* Algorithm development
* Coding and documentation
* Testing and debugging
* Maintenance

**FLOW CHART:**

Used for formulas

(a+b)2

Used for conditional statements like if, else if, while, for etc.

conditions

Used for start and stop

Start or stop

Used for i/p and o/p indicators

Print

**INTRODUCTION TO C**

* C language is a general-purpose high-level programming language which was introduced by Dennis M Ritchie in 1972 at bell laboratories in ANSI in America.
* C language was invented for finding language for UNIX operating system

It was named as c because it is the successor of B language .B language cannot able to find solutions for all problems that’s why C invented.

**FEATURES OF C:**

* Simple syntax
* Easy to learn
* Easy to code
* Procedure oriented programming language
* Memory management
* Speed

**COMMENTS**

There are two types of comments that which are available in c language those are:

1. Single line comment (//)
2. Multi line comment (/\*\_\_\_\*/)

**Memory representation of program consist of following sections**

1. Text segment
2. Initialized data segment
3. Uninitialized data segment
4. Stack
5. Heap

**Text segment:**

It contains program instructions for execution.

It is read only section of program and cannot be changed during execution.

It is placed in lower memory so it is not overwritten by heap or stack.

**Data Segment:**

It contains program instructions for execution.

It is read only section of program and cannot be changed during execution.

It is placed in lower memory adder so it’s not overwritten by heap or stack.

Two types:

* Initialized Data
* Uninitialized Data

**Initialized data:**

It contains global &static variables that are explicitly initialized.

Lifetime of variable in this section is throughout the execution of the program.

Variables in this can be changed.

It is also possible to change this in as read only memory.

Const char\* str=”hello world”;

int val;

int main()

{

Static int a=10;

Static int b;

Return 0;

}

**Uninitialized data section**:

It contains global and static data that is not initialized.

Lifetime of this variables are throughout the program execution.

**Heap Segment**:

Dynamic memory allocation that happens during program execution is stored in heap section.

Heap memory must manage by the program statements.

Malloc, realloc, calloc ,free are the functions used to manage heap memory.

If memory is not managed it leads to memory leak problems.

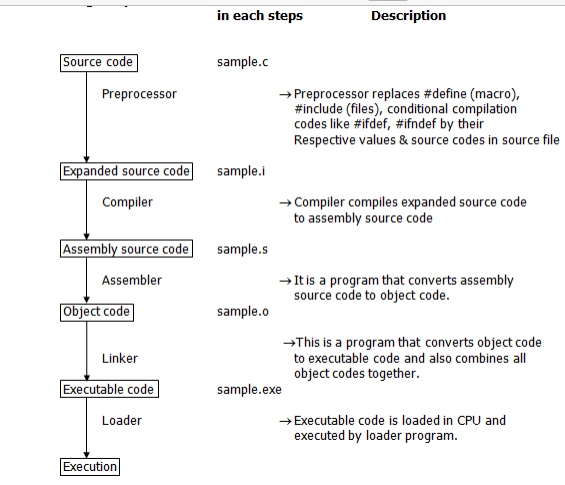
**Stack:**

It contains activation records or stack frames.

Stack frames are created for each function call &pushed in stack.

Stack frame is popped from stack when function returns.

**PROGRAM EXECUTION IN C**



**TOKENS IN C**

Tokens are nothing but smallest individual parts. And in C there are 6 types of tokens available.

1.Variables

2.Identifers

3.Keywords

4.Constants

5.Special Symbols

6.Strings

**Types of variables**:

**1.Camel case:**

Here each word except first word starts with capital letter.

myVariableName

**2.Pascal case:**

Each word starts with a capital letter.

MyVariableName

**3.Snake case:**

Here each word is separated by an underscore.

my\_varaible\_name.

**VARIABLES:**

* A variable is a name of memory location. It is used for store the data.
* variables are changeable we can change the value of the variable during the execution of the program.
* Variables must be declared before using them in program.
* If we don’t declare variables then we get error.

**Rules:**

* Variables must start with letters or underscore.
* Noo special symbols are used for declaring variables except underscore.
* Variables names cannot contain special names or spaces.
* Variable name cannot be keyword because keywords have special function in any language.
* 12,245 invalids.
* 10 20 30 invalids.

**IDENTIFIERS:**

Identifiers refer to the names of variables, constants, functions and arrays.

These are user-defined names is called Identifiers.

**Rules for an Identifier**

1. An Identifier can only have alphanumeric characters (a-z, A-Z, 0-9) and underscore (\_).

2. The first character of an identifier can only contain alphabet (a-z, A-Z) or underscore (\_).

3. Identifiers are also case sensitive in C.

For example, name and Name are two different identifiers in C.

4. Keywords are not allowed to be used as Identifiers.

5. No special characters, such as semicolon, period, whitespaces, slash or comma are permitted to be used in or as Identifier.

6. “C‟ compiler recognizes only the first 31 characters of an identifiers.

**Difference between identifier and a variable**

An identifier is a name used to identify a variable, function, array, structure, or any other user defined item.

A variable a storage location that holds a value.

Variables have different data types.

**KEYWORDS(C)**

* Keywords are reserved words in c language.
* There are 32 keywords which are available in c.
* Each keyword is meant to perform a special work.

Auto double goto signed unsigned

Break do if sizeof void

Case else int static volatile

Char enum long struct while

Const extern register switch

Continue float return typedef

Default for short union

**CONSTANTS IN C**

Constants mean fixed values it doesn’t change its values during execution time.

Constants are also called as literals.

There are 2 types of constants available in c:

1.Numeric Constant

2.Character constants

Numeric again classified into 2 types:

1. Real constants
2. Integer constants

Character constants again classified into 2 types:

1. Single character constants
2. Strig constants

**Integer constants:**

It is a numeric constant without an exponential part.

1. Decimal constant
2. Octal constant [must start with 0]
3. Hexadecimal constant [must start with 0X or 0x)

**Example:** #include<stdio.h>

int main()

{

Printf("\x41");//ascii value of A in hexadecimal

printf("\x52");//ascii value of R in hexadecimal

printf("\x69");//ascii value of i in hexadecimal

return 0;

}

In c programming octal constants are represented in 0 and hexadecimal constants are represented in 0x

For octal, decimal, hexadecimal no special characters are allowed.

There is one more constant in c i.e. escape characters or backslash charcter.

**Escape characters or backslash characters**:

For example, if we want to print the word **"I ♥ Cats"**with a universal character name and double quotes, we can use the following printf() statement-

printf("\"I \u2665 Cats\"");

Some characters are not part of the ASCII character set, such as accented letters or symbols. To use these characters in a string or a character literal, we need to use their octal or hexadecimal codes with a backslash (\).

\n newline

\r carriage return

\v vertical tab

\t tab

\a alert (beep sound)

\b backspace

**\r carriage returns**

**Example:**#include<stdio.h>

int main()

{

printf("hello\rworld!");

return 0;

}

**/t and /b Backspace and tab**

**Example:**

#include<stdio.h>

int main()

{

printf("hello\tworld!");//give spaces

printf("hello\b\b\b\bworld\n");//remove the letters that how many backspace we give

printf("errt\b");

return 0;

}

**/f Form feed**

The form feed character (ASCII code 12) is a control character that historically caused the printer to advance to the next page or form. In modern systems, it often represents a vertical spacing or page break in the output. But form feed is sometimes not worked in C.

**Example:**

#include <stdio.h>

int main() {

printf("I am \f Unstoppable");

return 0;

}

**Output**:I am

unstoppable

**/v Vertical Tab**

The vertical tab character adds vertical space between the two strings, i.e., the sections before and after the escape sequence \v.

**Example:**#include <stdio.h>

int main() {

printf("I am \v Unstoppable");

return 0;

}//

**Output:**

I am

Unstoppable

**\? Question Mark**

**Example:**#include <stdio.h>

int main(){

printf("Are you Unstoppable\?");

return 0;

}

**Null Character Escape Sequence in C (\0)**

The null character escape sequence is used to terminate a string and ignore the string segment following the null character.

**Example:**#include <stdio.h>

int main() {

printf("I am Unstoppable\0Continued");//i am unstopabble

return 0;

}

**Octal Value Escape Sequence in C (\nnn)**

We use the octal value escape sequence to insert/ print the character corresponding to that octal value

**Example:**#include <stdio.h>

int main() {

printf("I am\072 Unstoppable");//i am: unstoppable

return 0;

}

**Hexadecimal Value Escape Sequence in C (\xhh)**

It is possible to print all characters from the C character set in the output using their hex values in the hexadecimal escape sequence. The sequence consists of a backslash, followed by a zero, and then the hex value of the character.

**Example**: #include <stdio.h>

int main() {

printf("We are Unstoppable\x73\x21");// iam unstoppable!

return 0;

}

**Universal Character Name Escape Sequence in C (\uhhhh)**

The universal character name escape sequence denotes the Unicode points of characters in string literals. They have the form \uhhhh or \Uhhhh, with h corresponding to the hexadecimal digits.

**Example:** #include<stdio.h>

int main(){

printf("I am \u03A9 Unstoppable");

return 0;

}

**Constants**

There are 2 ways define constants in c

1.const keyword

2.#define preprocessors

**C const keyword:**

The const is used to define the constant in c programming

Const float pi=3.14

**#define**:

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

#define token value

#define pi 3.14

**DATA TYPES IN C**

There are two types of qualifiers in c

1. Size qualifier (short, Long)
2. Sign qualifier (unsigned, signed)

The range of values for signed data type is less than that of unsigned data type. Because in signed data type left most bit is used to represent sign, while in unsigned it is also used for represent value.

In c there are 4 types of data types are there:

1. Primitive or Primary or Built-in data type
2. Non primitive or user defined or secondary
3. Void
4. Enum

**Primitive Data Types**: predefined datatypes

There are:

1. Int
2. float
3. char

**Integer**

* 16 bits or 2 bytes
* -32768 to 32767
* %d or %i used for signed integers or integers

**Unsigned int**:

* 16 bits or 2 bytes
* 0 to 65535

**Short int:**

* 8 bits or 1 byte
* -128 to 127
* %d or %hi

**Unsigned short int:**

* 8 bits or 1 byte
* 0 to 255
* %d or %hu

**Long int:**

* 32 bits or 4 bytes
* -2147483648 to 2147483647
* %ld

**Unsigned long int**:

* 32 bits or 4 bytes
* 0 to 6,

**Float:**

* 32 bits 4 byte
* -3.4E-38 to 3.4E+38
* %f is used for float
* %e or %E is e for 3.6e, for 3.6E.

**Example**

#include<stdio.h>

int main()

{

float f=8.90000;

printf("%f\n",f);//8.900000

printf("%g\n",f);//8.9

printf("%G\n",f);//8.9 it doesn’t give extra zeros what value we give that only it printed and also up to 5 letters only

printf("%e",f);//8.999E+…

return 0;

}

Following are other examples of various types of Integer literals:

An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

* 85 decimals
* 0213 octal
* 0x4b hexadecimal
* 30 int
* 30u unsigned int
* 30l long
* 30ul unsigned long

**Double:**

* 64 bits or 8 bytes
* 1.7E-308 to 1.7E+308
* %f or %lf

**Long double:**

* 80 bits or 10 bytes
* 3.4E-4932 to 1.1E+4932
* %Lf

**Character Type:**

* 8bits or 1 byte
* -128 to 127
* %c

**Unsinged char**

* 8bits or 1 byte
* 0 to 255
* %c

**Non primitive or derived data type**:

These are derived by the primitive data types.

E.g. structures, unions, pointers, arrays, strings

**Enum:**

An Enum is a keyword it is a user defined data type all properties of integer are applied on enumeration data type so size of the enumerator data type is 2byte.it works like a integer

Using Enum we can create sequence of integer constant value.

Enum tagname{value1value2value3------}

Here Enum is user defined data type

Tagname is our own variable

Value1,value2,----set of Enum values.

Enum week (sun, mon, tue, wed, thr, fri, sat);

It is start with zero by default and value is incremented by 1 for the sequential identifiers in the list.

If constant one value is not initialized than default sequence will be start from zero

#include<stdio.h>

Enum sbc{x,y,z};

Int main ()

{

Int a;

A=x+y+z;

Printf(“sum %d”,a);

Return 0;

}

Sum=3

**TYPE CONVERSION OR CASTING:**

Converting one data type to another data type.

There are two types of those are:

1. Implicit (widening)
2. Explicit (Narrowing)

**Implicit:**

* Also called as automatic type conversion.
* Here lower data type is assigned to higher data type.

//automatic typecasting

/\*#include<stdio.h>

int main()

{

char a='5';

int b=9;

int result=a+b;

printf("%d",result);

return 0;

}\*/

**Explicit:**

* These is useful when higher data type is converted to lower data type
* Here it doesn’t happen automatically ‘here we have to mention what data type we have to convert
* Because here we are converting the Higer data type to lower data type so higher can take more bits than lower data type.so here wastage happens that’s why we have to mention.

Syntax: (type\_name) expression;

**Example:**

#include<stdio.h>

int main()

{

int a=10,b=3;

float c=a/b;

printf("%f",c);//3.0000

return 0;

}

/\*#include<stdio.h>

int main ()

{

int a=10, b=3;

float c=(float)a/b;//3.33333//this one is the correct explicit typecasting

printf("%f",c);

return 0;

}\*/

/\*#include<stdio.h>

int main()

{

int a=10,b=3;

float c=(float)(a/b);

printf("%f",c);

return 0;

**Difference between type casting and type conversion**:

Type casting and type conversions are related concepts in programming, but they are not exactly same.

Type casting:

It is the process of explicitly changing.

It is often used when the programmer knows that the conversion is safe and necessary for a particular operation.

Type conversion:

Also known as type coercion, it is implicitly done, which is handled by language.

**Storage classes:**

Every variable in c programming has 2 properties

Type& storage class

Type: type refers to the what type of data type it is.

Whereas storage class determines how long it stays in existence.

Storage classes makes our program fast and efficient.

* These are used to determine the lifetime, visibility, memory location. And initial values of variable.
* Initial value what would be the initial value.
* Lifetime it is the time between creation and distribution of a variable. Or how long the variable would exist.
* There are four types of storage classes available in c language

1. Auto
2. Register
3. Static
4. Extern

**Auto:**

* **Storage**: stack frame of the corresponding function
* **Initial value:** garbage values
* **Scope**: function/block scope
* **Lifetime**: memory gets created when function is invoked or call and memory gets destroyed when function returns.
* Every local variable is automatic in c language.

**Example:**

#include<stdio.h>

void func(void);

int main()

{

func();//x=2,y=5

func();//x=2,y=5

func(); //x=2,y=5

}

void func(void)

{

int x=2, y=5;//here x and y values created

printf("x=%d,y=%d",x,y);

x++, y++;//here the variable gets destroyed

}

**Register:**

* The variables defined in the register is allocated the memory in the CPU register.
* Register is one of the parts in CPU.
* In CPU register is not available then compiler will treat them as automatic variables.
* Register variables don’t have memory addressed so we can’t apply address operator (&) to them.
* We can declare only few variables as register.
* It is better to use most commonly used variables.
* The initial default value for the register is 0.
* The access time of the register variable is faster than automatic variables.
* The register should only use for variables that require quick access such as counters. It should be noted that defining register does not mean that the variable will be stored in register depending on hardware and implementation restrictions.
* Here the variable is stored in register maximum size of variable=maximum size of the register.
* The register storage classes can be applied to only local variables
* The scope, lifetime and initial values of register variables are same as that of automatic variables.

**Syntax**: Register int varaibale\_name;

**External or global:**

* Declared outside of the function and are accessible to all functions in program i.e. we can be accessed between two different files.
* These are global variables
* Extern variables are declared using the extern keyword.
* **Storage:** RAM (i.e. data segment)
* **Scope**: multiple files.
* **lifetime**: till end of the program.
* **Initial value**: zero.
* Extern int variable\_name.
* Extern is used whenever we have multiple numbers of files while we are trying to define any global variable or function. That will be used in various other files too then we will use the extern in anther file for providing the refence of the defined function or variable.

**Static:**

**Two types:**

* Local static variables
* Global static variables

**Local static variables**:

The lifetime of a static variable is same as automatic variable.

Static int x=6;/\*valid\*/

Int x=8;

Static int y=x;/\*invalid\*/

**Storage:** Data/BSS segment

**Initial value**: Zero

**Scope**: function scope

**Lifetime**: memory is created when program starts execution and memory is destroyed as soon as when program terminates

**Global static variables**:

If a local variable is declared as static it remains alive throughout the program

In the case of global variables, the static specifier is not used to extend the lifetime since global variables have already a lifetime equal to the life of program

Static specifier is used for information hiding

**Storage**: Data/BSS segment

**Initial value:** zero

**Scope:** file scope

**Lifetime**: memory is created when program starts execution and memory is destroyed as soon as when program terminates.

**Operators in c**

Operators are divided into 3 types:

* Unary
* Binary
* Ternary

operators are nothing but that which acts upon on operands.

In c there are 8 types of operators are there:

* **Ari thematic operators** (+, -, \*, /, %)
* **Relational operators** (==, =,>, <, <=,>=)
* **Shift operators** (left shift and right shift)

1. **Left shift (<<):** The left shift operator moves the bits of the first operand the left by the number of places of the second argument.
2. **Right shift operator (>>):** The right shift operator shifts the bits of the first operand to the right but the number of positions specified by the second operand

* **Logical operators**: perform logical operations and returns true or false (AND (&&), OR (||), NOT(!))
* **Bitwise operators**: It only operate son the single bits (&, |, ^, ~)

Twist in bitwise complement operator in C Programming The bitwise complement of 35 (~35) is -36 instead of 220, but why?

For any integer n, bitwise complement of n will be -(n+1).

**Bit & Operation of 12 and 25**

00001100

00011001

00001000 = 8 (In decimal)

**Bitwise OR Operation of 12 and 25**

00001100

| 00011001

\_\_\_\_\_\_\_\_

00011101 = 29 (In decimal)

**Bitwise XOR Operation of 12 and 25**

00001100

| 00011001

\_\_\_\_\_\_\_\_

00010101 = 21 (In decimal)

* **Ternary or conditional operator** it is same as if else condition in conditional statements.

conditional operator checks the condition and executes the statement depending on the condition. A conditional operator is a ternary operator, that is, it works on 3 operands. Conditional operator consists of two symbols.

1: question mark (?).

2: colon (:).

Syntax: condition? exp1: exp2;

It first evaluates the condition, if it is true (non-zero) then the “exp1” is evaluated, if the condition is false (zero) then the “exp2” is evaluated.

* **Assignment operator** assignment operators are used to assign the values to the variables (=, +=, -=, \*=, /=, %=,)
* **Miscellaneous operator** (The size of and the comma operator comes under the miscellaneous operator)

Comma operator evaluate multiple expressions and gives the last expression.

**Increment operator:**

Whenever more than on format specifier is directly or indirectly linked related to same variable then we need to evaluate the expression from right to left.

#include<stdio.h>

int main ()

{

Int i=1;

Printf(“%d%d%d”,i,++i,++i);

}

o/p:3 3 1

We cannot use increment operator on the constant values because increment operator operates on only variables. It increments the value of the variable by 1 and stores the incremented value back to the variable

**CONTROL STATEMENTS OR CONDITIONAL STATEMENTS OR SELECTION STATEMENTS IN C**

1. if
2. If else
3. Else if
4. Nested if

**If**

if(expression) {

//code to be executed

}

**Example:**

#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;

}

**If else:**

if(expression) {

//code to be executed if condition is true

}

else{

//code to be executed if condition is false

}

**Example:**

#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);

}

else{

printf(%d is odd number, number);

}

return 0;

}

**Else if ladder:**

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

}

**Example:**

#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;

}

**SWITCH:**

The switch statement in C is an alternate to if-else-if ladder statement which allows us to execute multiple operations for the different possibles 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.

switch(expression) {

case value1:

 //code to be executed;

break; //optional

case value2:

 //code to be executed;

 break; //optional

......

default:

**Rules**

* The *switch expression* must be of an integer or character type.
* The *case value* must be an integer or character constant.
* The *case value* can be used only inside the switch statement.
* The *break statement* in switch case is not must. It is optional. If there is no break statement found in 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. Even if the case constants of the inner outer switch contain common values, no conflicts will arise.

**Example:**

#include <stdio.h>

int main() {

int num = 2;

switch (num {

case 1:

printf(Value is 1\n);

break;

case 2:

printf("Value is 2\n");

break;

case 3:

printf("Value is 3\n");

break;

default:

printf("Value is not 1, 2, or 3\n");

break;

}

   return 0;

}

**LOOPS:**

The looping can be defined as repeating the same process multiple times until a specific condition satisfies. The looping simplifies the complex problems into the easy ones. It enables us to alter the flow of the program so that instead of writing the same code again and again, we can repeat the same code for a finite number of times.’

There are mainly two types of loops are there in c

Entry controlled: Here in entry controlled the statements or conditions is verified, at that beginning of the program.

1. **While loop or sentinel-controlled loop**

variable initialization;

while (condition) {statements;

variable increment or decrement;

}

The other two loops i.e. while and do while loops are more suitable in the situations where it is not known before hand when the loop will terminate.

**Examples:**

#include<stdio.h>

int main()

{

while(1)

{

printf("hello world");

}

return 0;

}

#include<stdio.h>

int main()

{

while(-10)

{

printf("hello");

}

return 0;

}

#include<stdio.h>

int main()

{

while('a')

{

printf("hello world");

}

}

1. **For loop or counter controlled loop**

for (initialization; condition; increment/decrement)

{

Statements;

}

The for loop is appropriate when we know in advance how many times the loop will be executed.

**Examples:**

#include<stdio.h>

int main()

{

int i;

for(i=0;i++<5;)

printf("%d",i);

return 0;

}

#include<stdio.h>

int main()

{

int x,y;

for(x=1,y=1;x<=5,y<=5;x++,y++)

{

printf("%d\*%d=%d",x,y,x\*y);

}

return 0;

**Exit controlled**: whereas in exit controlled the condition is verified after at the end of the program.

**Do while**

Here in the do while condition the code run at least one time even the condition is satisfied or not is not a matter.

Whenever semicolon is placed at the end of the while loop this means that the body value never gets incremented.

Syntax:

variable initialization;

do

{

statements;

variable increment or decrement;

}

while (condition);

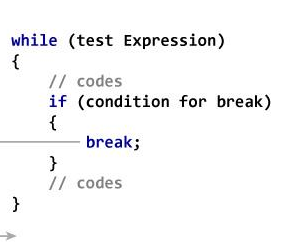
**JUMP OR LOOP CONTROL OR UNCONDITIONAL STATEMENTS:**

* Break
* Continue
* Goto

**Break:**

By using these we can terminate the further execution of the program.

Syntax: break;



**Continue:**

These is the fully opposite to break statement.

It is used for continuing the next iteration of loop statements.

When continue statement occurs in the loop it does not terminate but skips the statement after the continue statement.

**Example:**

#include <stdio.h>

int main(){

int i=1;//initializing a local variable

//starting a loop from 1 to 10

for(i=1;i<=10;i++){

if(i==5){//if value of i is equal to 5, it will continue the loop

continue;

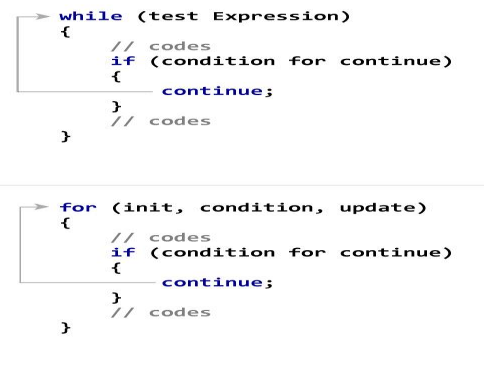
}

printf("%d \n",i);

}//end of for loop

return 0;

}



**Goto:**

It does not require any condition in the loop.

This statement passes control anywhere in the program i.e. control is transferred to another part of program without testing any condition.

The goto statement requires a label to identify the place to move to execution label is valid variable name and must tended with colon.

Goto label;

-------

------- forward branching

Label:

Label:

------ backward branching

-------

Goto label;

Use of goto statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program making the program hard to understand and hard to modify.

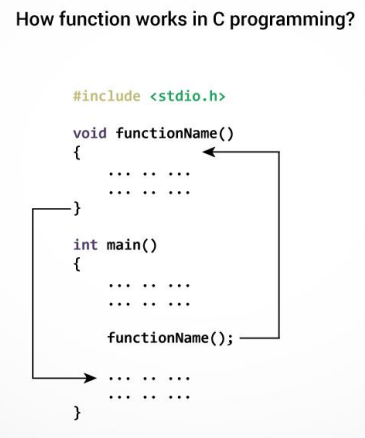
Here label can be any plain text except c keyword and it can be set anywhere in the program

**FUNCTIONS OR MODULE OR SUBROUTINE**

A function is a finite set of instruction that together perform a specific task.

A large program is divided into basic building blocks called function

Function is also called as module, procedure, subroutine.



3-types:

1. Function declaration or function prototype
2. Function call
3. Function definition

The main advantage of the function is code reusability i.e. if we want to perform same task repeatedly in another program then function can be denied as many numbers of times.

**Function declaration**:

Return\_type function-name (argument list);

Function call

Function name (arguments list);

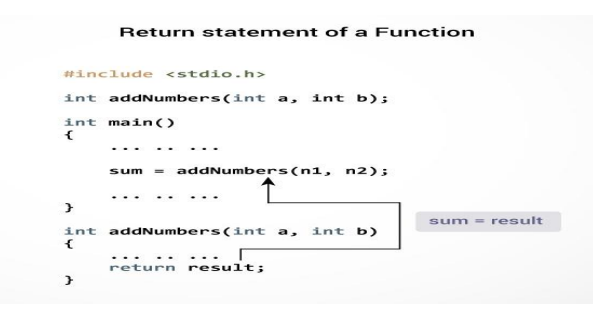
**Function definition:**

Return type function name (arguments list)

{

Body of function

}



In c there are two types of functions available:

1. User defined
2. Pre-defined function

**Types of user defined function:**

1. Function with arguments and with return value
2. Function with no arguments and no return value
3. Function with arguments and without return value
4. Function without arguments and with return value.

**Advantages of the functions:**

1. code reusability
2. Less coding

**Function with no arguments and no return value:**

Void function-name();

Function-name();

Void function-name()

{

Body of function

}

**Function with no arguments and return value**:

Int function-name();

Function name();

Int function-name()

{

Statements;

Return value;

}

**Function with arguments but no return value**:

Void function-name(int);

Function-name ();

Void function-name (int variable)

{

Statements;

}

**Function with arguments and with return value**:

Int function name(int);

Function name(variable);

Int functionate (int variable)

{

Statements;

Return type;

}

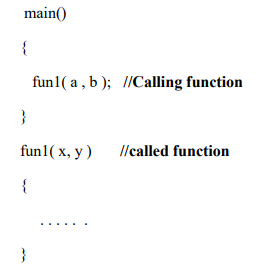
Parameters are not important in function declaration, but their type is important.

**Arguments:**

That which is passed into a function by its caller at the i.e. variable is declared at that time of function calling

**Parameters:**

That which is received by the function i. e. a variable held or received at the time of function definition.



Parameters

There are two types

* Actual parameter
* Formal parameter

**Actual parameter:**

That which are in calling function are called actual parameter

That which are used in called function are called as formal parameter.

**Formal parameter** is also called as dummy parameter

For large programs we preferred to use complier function

 functionName([arg1, arg2, ...argN]){

//code to be executed

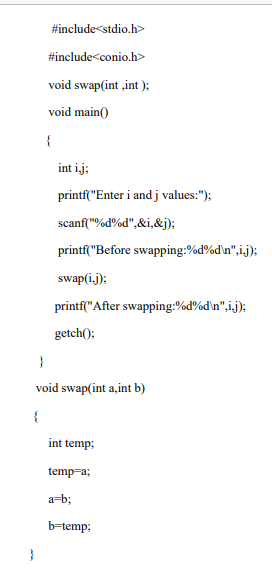
}

There are two ways which we can pass the arguments to the function such as

1. call by value or pass by value
2. call by reference or pass by reference

**call by value:**

In the call by value copy of the actual argument is passed to the formal arguments and operation is done on formal argument. In call by value, value being passed to the function is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only. It will not change the value of variable inside the caller method such as main(). when the function is called by call by value method it doesn’t affect content of actual arguments changes made in the formal arguments are local to the block of called functions when control back to calling functions the change will vanish.



o/p: enter the values:10 20

before swapping 10 20

after swapping 10 20

**Call by reference:**

Instead of passing the value to the variable address or reference is passed and the function operate on address of the variable rather than value

Actual parameters can be constants, variables, or expressions

C=add(a,b);

C=add(a+5,b);

C=add(10,20);

Formal parameters should be only variables, expressions and constant are not allowed.

Actual parameters send values to the formal parameters formal parameters receive values from the actual parameter

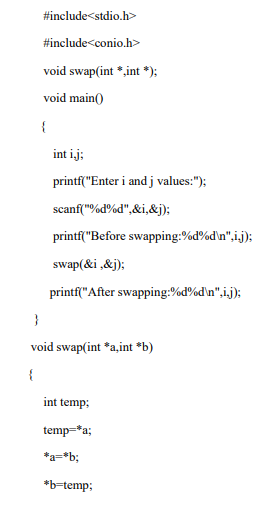
Here in call by value being passed is stored in the stack memory location if change the value of function parameter it is changed for the current function only

In call by reference function is called with address of actual parameter and the formal parameters receive the addresses of the actual parameter now the formal parameter doesn’t contain the values instead of the it contains an address it is called as pointer variable.

Using pointer variable, the value of the actual parameter gets changed

In call by reference the memory locations are same

Whereas in call by value the memory locations are different.



o/p: enter the elements 10 20

before swapping 10 20

after swapping 20 10

**ARRAYS**

Array is a collection of similar data type and stored in continuous memory.

* The lowest address corresponds to the first element and highest element address to the last element.
* Array of character is string.
* Each data item in array is called an element. And each element in array is unique and located in separated memory location.
* Each of elements of an array share same variable name but each element is having different subscripts.
* Array can be single or multidimensional number of subscripts determines the type of the dimension.
* It can be used to represent multiple data structures like linked lists, stacks, queues, trees, graphs etc.
* We must know in advance that how many elements we have to store.
* Array is static structure i.e. fixed size the memory which is allocated we are not able to increase or decrease.
* Symbolic also be used to specify the size of the array

#define size 10;

* Index of array starts from 0 and end with -1.
* If during the initialization of number, the initializers is less than size of the array then all the remaining elements assigned to be zero.
* If the number of elements is more than the size given in brackets then the compiler will show an error.

**Need of array**

Suppose you have to store marks of 50 students, one way to do this is allotting 50 variables. So, it will be typical and hard to manage. For example, we cannot access the value of these variables with only 1 or 2 lines of code.

Types:

1. Single Dimension
2. Two Dimension
3. Multi Dimension

**Single- or One-Dimensional Array**

* It is used to represent and store data in a linear form.
* Array having only one subscript is called as one-dimensional value.

**Two-dimensional array**

* It is used to represent matrix.
* No of bytes required for 2 dimensional=no of rows \* no of columns \* size of data type.

**Examples:** **//Compile time array initialization**

#include<stdio.h>

#include<conio.h>

void main()

{

int i;

int arr[]={2,3,4};

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

printf("%d\t",arr[i]);

}

getch();

}\*

**//array initilization**

#include<stdio.h>

#include<conio.h>

void main(){

int i=0;

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

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

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

}

getch();

}

**//runtime array initialization**

#include<stdio.h>

#include<conio.h>

void main()

{

int arr[4];

int i, j;

printf("Enter array element");

for(i=0;i<4;i++)

{

scanf("%d",&arr[i]); //Run time array initialization

}

for(j=0;j<4;j++)

{

printf("%d\n",arr[j]);

}

getch();

}

#include <stdio.h>

#include <conio.h>

void main(){

int i=0,j=0;

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

clrscr();

//traversing 2D array

for(i=0;i<4;i++){

for(j=0;j<3;j++){

printf("arr[%d] [%d] = %d \n",i,j,arr[i][j]);

}//end of j

}//end of i

getch();

}

**addition of matrices**

#include<stdio.h>

#include<conio.h>

void main()

{

int a[25][25],b[25][25],c[25][25],i,j,m,n;

clrscr();

printf("enter the rows and colums of two matrics:\n");

scanf("%d%d",&m,&n);

printf("\nenter the elements of A matrics");

for(i=0;i<m;i++)

{

for(j=0;j<n;j++)

scanf("\t%d",&a[i][j]);

}

printf("\nenter the elements of B matrics");

for(i=0;i<m;i++)

{

for(j=0;j<n;j++)

scanf("\t%d",&b[i][j]);

}

printf("\nThe elements of A matrics");

for(i=0;i<m;i++)

{

printf("\n");

for(j=0;j<n;j++)

printf("\t%d",a[i][j]);

}

printf("\nThe elements of B matrics");

for(i=0;i<m;i++)

{

printf("\n");

for(j=0;j<n;j++)

printf("\t%d",b[i][j]);

}

printf("\nThe additon of two matrics");

for(i=0;i<m;i++)

{

printf("\n");

for(j=0;j<n;j++)

{

c[i][j]=a[i][j]+b[i][j];

printf("\t%d",c[i][j]);

}

}

getch();

}\*/

**//multidimensional array**

/\*#include <stdio.h>

int main()

{

// this array can store 12 elements

int i, j, k, test[2][3][2];

printf("Enter 12 values: \n");

for(i = 0; i < 2; ++i) {

for (j = 0; j < 3; ++j) {

for(k = 0; k < 2; ++k ) {

scanf("%d", &test[i][j][k]);

}

}

}

// Displaying values with proper index.

printf("\displaying values:\n");

for(i = 0; i < 2; ++i) {

for (j = 0; j < 3; ++j) {

for(k = 0; k < 2; ++k ) {

printf("test[%d][%d][%d] = %d\n", i, j, k, test[i][j][k]);

}

}

}

return 0;

}\*/

**Structures:**

* Structures used to store for the different data types.
* Here we will use struct keyword.

**Syntax:**

Struct taganme

{

Datatype member1;

Datatype member2;

Datatype member3;

}

E.g:

Struct student {

Char name [64];

Char course [12];

Int age;

Int year;

};

**Example Programs for structure**

#include <stdio.h>

#include <string.h>

struct Books

{

char title[50];

char author[50];

char subject[100];

int book\_id;

};

int main( )

{

struct Books Book1; // Declare Book1 of type Book

struct Books Book2; //Declare Book2 of type Book

//book 1 specification

strcpy( Book1.title, "C Programming");

strcpy( Book1.author, "Nuha Ali");

strcpy( Book1.subject, "C Programming Tutorial");

Book1.book\_id = 6495407;

//book 2 specification

strcpy( Book2.title, "Telecom Billing");

strcpy( Book2.author, "Zara Ali");

strcpy( Book2.subject, "Telecom Billing Tutorial");

Book2.book\_id = 6495700;

// print Book1 info

printf( "Book 1 title : %s\n", Book1.title);

printf( "Book 1 author : %s\n", Book1.author);

printf( "Book 1 subject : %s\n", Book1.subject);

printf( "Book 1 book\_id : %d\n", Book1.book\_id);

// print Book2 info

printf( "Book 2 title : %s\n", Book2.title);

printf( "Book 2 author : %s\n", Book2.author);

printf( "Book 2 subject : %s\n", Book2.subject);

printf( "Book 2 book\_id : %d\n", Book2.book\_id);

return 0;

}

* Each variable declared in structure is called as member.
* Name given to the structure is called tag.
* Members structures are enclosed within the opening and closing braces.
* Like primary variables structure variables can also be initialized when they are declared.
* Structure templates can be locally or globally. If it is local then it can be used within that function
* If it is global, it can be used by all other functions of program.
* If initializer is less than no of structure variables. Automatically rest of values taken as zero.
* Dot operator is used to access elements.

Structure variable;

S1.Name[];

S1.roll;

S1.age;

* Elements of structure are stored in contiguous memory locations.
* Value of structure variable can be assigned to another variable of same data type using assignment operator.
* Unary, relational, arithmetic, bitwise operators are not allowed within structure variable.
* Size of structure can be found by using size of () of operator

Size of(s1);

Pointers to Structures You can define pointers to structures in very similar way as you define pointer to any other variable as follows:

struct Books \*struct\_pointer;

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the & operator before the structure's name as follows:

struct\_pointer = &Book1;

To access the members of a structure using a pointer to that structure, you must use the -> operator as follows:

struct\_pointer->title;

Let us re-write above example using structure pointer, hope this will be easy for you to understand the concept:

#include<string.h>

struct Books

{

char title[50];

char author[50];

char subject[100];

int book\_id;

};

function declaration

void printBook( struct Books \*book );

int main()

{

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

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

/\* book 1 specification \*/

strcpy( Book1.title, "C Programming");

strcpy( Book1.author, "Nuha Ali");

strcpy( Book1.subject, "C Programming Tutorial");

Book1.book\_id = 6495407;

/\* book 2 specification \*/

strcpy( Book2.title, "Telecom Billing");

strcpy( Book2.author, "Zara Ali");

strcpy( Book2.subject, "Telecom Billing Tutorial");

Book2.book\_id = 6495700;

/\* print Book1 info by passing address of Book1 \*/

printBook( &Book1 );

/\* print Book2 info by passing address of Book2 \*/

printBook( &Book2 );

return 0;

}

void printBook( struct Books \*book )

{

printf( "Book title:%s\n", book->title);

printf( "Book author:%s\n", book->author);

printf( "Book subject:%s\n", book->subject);

printf( "Book book\_id:%d\n", book->book\_id);

}

**UNION:**

* Unions is almost same as structures. Except one key difference structures allocate enough space store all their members, whereas unions can only hold one member value at a time. Due to this only one member can store data at the given instance.
* Here we will use union keyword.
* We use . operator to access members of a union and to access pointer variables we use the -> operator.
* Keep in mind that we have to always end the union declaration with a semicolon.
* The size of the union will always be equal to the size of the largest member of the array. All these sized elements can store the data in the same space without any overflow.

**Example:**

#include <stdio.h>

#include <string.h>

union Data

{

int i;

float f;

char str[20];

};

int main( )

{

union Data data;

printf( "Memory size occupied by data : %d\n", sizeof(data));

return 0;

}\*/

#include <stdio.h>

union Data

{

int i;

float f;

char str[20];

};

int main( )

{

union Data data;

data.i = 10;

printf( "data.i : %d\n", data.i);

data.f = 220.5;

printf( "data.f : %f\n", data.f);

strcpy( data.str, "C Programming");

printf( "data.str : %s\n", data.str);

return 0;

}

//access union members

/\*#include <stdio.h>

#include <string.h>

union Data

{

int i;

float f;

char str[20];

};

int main( )

{

union Data data;

data.i = 10;

data.f = 220.5;

strcpy( data.str, "C Programming");

printf( "data.i : %d\n", data.i);

printf( "data.f : %f\n", data.f);

printf( "data.str : %s\n", data.str);

return 0;

}\*/

**STRINGS**

Strings are nothing but array of characters ended with \0. It indicates end of the string.

Strings can be declared and initialized by using 2 ways:

* By using arrays
* By using pointers

Datatype \*array\_name;

Char \*variable;

Dataytype arraryname[index];

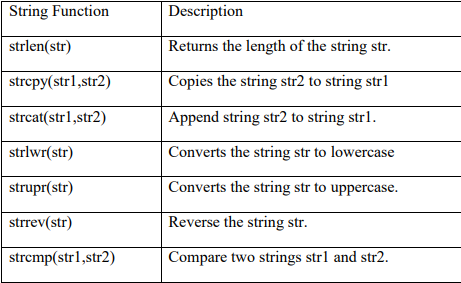
Char name [10];

Char b[9]={„C‟,‟O‟,‟M‟,‟P‟,‟U‟,‟T‟,‟E‟,‟R‟};

char b[]={„C‟,‟O‟,‟M‟,‟P‟,‟U‟,‟T‟,‟E‟,‟R‟};

char b[ ] = “COMPUTER”;

**STRING LIBRARY FUNCTIONS**:



All these functions must be in header file string.h

**strlen(string)** –String Length: This function is used to count and return the number of characters present in a string.

Syntax : var=strlen(string);

**Example:**#include<stdio.h>

#include<string.h>

int main()

{

char name[]="rupa";

int len1;

len1=strlen(name);

printf("the length of the string %s is %d",name,len1);

return 0;

}\*/

**String Copy:** This function is used to copy the contents of one string to another string. Syntax: strcpy(string1, string2);

Where string1: is the destination string.

string 2: is the source string. i.e the contents of string2 is assigned to the contents of string1.

**Example:**#include<stdio.h>

#include<string.h>

int main()

{

char str1[]="rupa";

char str2[10];

strcpy(str2,str1);

printf("the string1 is %s\n",str1);

printf("the string2 is %s\n",str2);

strcpy(str2,str1+2);

printf("the string is %s\n",str1);

printf("the string2 is %s\n",str2);

}\*/

/\*#include<stdio.h>

int main()

{

char str1[10],str2[10];

int index;

printf("enter the string");

scanf("%s",str1);

for(index=0;str1[index]!='\0';index++)

str2[index]=str1[index];

printf("string2 is %s\n",str2);

return 0;

}\*/

**strlwr(string)** – String LowerCase : This function is used to converts upper case letters of the string in to lower case letters.

**Example** #include<stdio.h>

#include<string.h>

int main()

{

char str[]="RUPA";

A=strlwr(str);

printf("the lowercase is %s",A);

return 0;

**strupr(string) –** String UpperCase : This function is used to converts lower case letters of the string in to upper case letters.

**Example:** #include<stdio.h>

#include<string.h>

int main(){

char str[]="rupa";

strupr(str);

printf("the string is %s",str);

return 0;

}

**strcmp(string1, string2)** – String Comparison : This function is used to compares two strings to find out whether they are same or different. If two strings are compared character by character until the end of one of the strings is reached. If the two strings are same strcmp() returns a value zero.

Syntax: strcmp(string1, string2);

Return a value

<0 when s1<s2

=0 when s1=s2

>0when s1>s2

The exact value returned in case of dissimilar strings is not defined. We only know that if s1<s2 then a negative value will be returned and if s1>s2 then a positive value will be returned.

**Example:**

#include<stdio.h>

#include<string.h>

int main()

{

char str1[10], str2[10];

printf("enter two strings");

scanf("%s%s",&str1,&str2);

if(strcmp(str1,str2)==0)

{

printf("strings are same");

}

else

{

printf("strings are not same");

}

return 0;

}

**strcat(string1,string2)** – String Concatenation : This function is used to concatenate or combine, two strings together and forms a new concatenated string.

Syntax: strcat(sting1,string2);

Where string1 : is the first string1. string2 : is the second string2 when the above function is executed, string2 is combined with string1 and it removes the null character (\0) of string1 and places string2 from there.

**Example:**#include<stdio.h>

#include<string.h>

int main()

{

char str1[10]="rupa";

char str2[]="ece";

strcat(str1,str2);

printf("%s",str1);

return 0;

}

**strrev(string)** - String Reverse: This function is used to reverse a string. This function takes only one argument and return one argument.

**Example:**#include<stdio.h>

#include<string.h>

int main()

{

char str[20];

printf("enter the string");

scanf("%s",str);

printf("the string reversed is %s",strrev(str));

return 0;

}

**Pointers:**

pointer is a variable that stores/hold address of another variable of same data type.

A pointer is derived data type in C.

**Benefits of using pointer**:

It allows C to support dynamic memory location.

It reduces length and program execution of time.

**NULL POINTER**

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

Syntax:

Data\_type \*pointer\_variable\_name;

Int \*P; or int\* p;//here \* means here the address belongs to the data type int.

Void data type is work for all the data type but it is rarely used.

**Pointer initialization is the process of assigning address of a variable contains address of a variable of same data type**.

Int a=10;

Int \*ptr;//pointer declaration

Ptr=&a;//pointer initialization

Int \*ptr=&a;//both pointer and array initialization

Normal values store the value, whereas pointer a variable point to address of the variable.

The content of the pointer i.e. address is always whole number.

&symbol is used to get the address of the variable

\*Is used to get the value of the variable that pointer is pointing to

If pointer is assigned to NULL then it is called as null pointer

We can able to perform subtraction on pointer variables, but we can’t do addition, multiplication divisions

**Example:**

Int a,\*p;

A=10

P=&a;

Printf(“%d”,\*P)//this will print the value

Printf(“%u”,&a);//this will print the address

Printf(“%u”,p);//this will print the address

Printf(“%d”,\*&a);//points value

Return 0;

}

**Example:**

#include stdio.h>

int main ()’

{

int var = 20;/\* actual variable declaration \*/

int \*ip; /\* pointer variable declaration \*/

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

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

printf("Address stored in ip variable: %x\n", ip );/\* access the value using the pointer \*/

printf("Value of \*ip variable: %d\n", \*ip );

return 0;

}

**Example:**

#include<stdio.h>

const int MAX = 3;

int main ()

{

int var[] = {10, 100, 200};

int i, \*ptr;

/\* let us have array address in pointer \*/

ptr = var;

for ( i = 0; i < MAX; i++)

{

printf("Address of var[%d] = %x\n", i, ptr );

printf("Value of var[%d] = %d\n", i, \*ptr );

/\* move to the next location \*/

ptr++;

}

return 0;

}

When the above code is compiled and executed, it produces result something as follows:

Address of var[0] = bf882b30

Value of var[0] = 10

Address of var[1] = bf882b34

Value of var[1] = 100

Address of var[2] = bf882b38

Value of var[2] = 200

**Decrementing a pointer**:

#include stdio.h>

const int MAX = 3;

int main ()

{

int var[] = {10, 100, 200};

int i, \*ptr; /\* let us have array address in pointer \*/

ptr = &var[MAX-1];

for ( i = MAX; i > 0; i--)

{

printf("Address of var[%d] = %x\n", i, ptr );

printf("Value of var[%d] = %d\n", i, \*ptr );

/\* move to the previous location \*/

ptr--;

}

return 0;

}

**Pointer comparison:**

#include<stdio.h>

const int MAX = 3;

int main ()

{

int var[] = {10, 100, 200};

int i, \*ptr;

/\* let us have address of the first element in pointer \*/

ptr = var; i = 0;

while ( ptr <= &var[MAX - 1] )

{

printf("Address of var[%d] = %x\n", i, ptr );

printf("Value of var[%d] = %d\n", i, \*ptr ); /\* point to the previous location \*/

ptr++;

i++;

}

return 0;

}

When the above code is compiled and executed, it produces result something as follows:

Address of var[0] = bfdbcb20

Value of var[0] = 10

Address of var[1] = bfdbcb24

Value of var[1] = 100

Address of var[2] = bfdbcb28

Value of var[2] = 200

**Dynamic memory allocation:**

It means that the memory will be allocated at the run time.

Memory management functions are used for allocating and freeing memory.

These functions are defined as stdlib.h

Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file

1. Malloc
2. Calloc
3. Realloc
4. Free

**Static:**

Memory allocated at the compile time

Memory can’t be executing while executing the program

Used in arrays.

**Dynamic:**

Memory allocates run time

Memory can be increased while executing program

Used in a linked list

**Methods:**

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

Global variables and static variables are stored in permanent storage area whereas local variables get stored in area called stack

The memory space between these two regions is known as heap area this region is used for dynamic memory allocation, during execution the size of the heap keep changes.

**Malloc():**

It stands for memory allocation.

The malloc function allocates single block of the requested memory at runtime this function reserves a block of memory of given size and returns a pointer of type void.

This means that he we can assign it to any type of pointer using type casting

It doesn’t initialize memory at execution time so it is having garbage value initially.

If it fails to allocate the enough space then it will return a NULL pointer.

Syntax;

Ptr=(cast\_type\*)mallaoc(byte\_size)(here cast type we are using \* because here malloc function returns the address)

Int \*x;

X=(int x)malloc(100\*sizeof(int));(here for 100 variables wea re going to use in our program)

Free(x);(in normal programming the memory will be allocated and automatically deallocated whereas in dynamic memory allocation we have to clear the space using free.

This statement will allocate either 200 or 400bytes according to the size of the 2 or 4 bytes respectively.

And pointers point to the address of the first byte of memory.

It is mostly used for structures.

It default value is garbage value.

Here if it finds to difficult to assign memory then it will return null.

**Example:**

#include <stdio.h>

#include <stdlib.h>

int main()

{

int num, i, \*ptr, sum = 0;

printf("Enter number of elements: ");

scanf("%d", &num);

ptr = (int\*) malloc(num \* sizeof(int)); //memory allocated using malloc

if(ptr == NULL)

{

printf("Error! memory not allocated.");

exit(0);

}

printf("Enter elements of array: ");

for(i = 0; i < num; ++i)

{

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

sum += \*(ptr + i);

}

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

free(ptr);

return 0;

}

**Calloc:**

Stands for contiguous allocation

Calloc is another memory allocation function that is used for allocating memory at runtime

Calloc function is normally used for allocating memory to derived data types like arrays .

The calloc function allocates multiple block pf requested memory

It initializes or sets all values to zero.

If it fails to locate enough space it returns null pointer

The only difference between the calloc and malloc is malloc single block of memory whereas calloc multiple block of memory

Syntax:

Ptr=(cast\_tyepe\*)calloc(number,elemnt\_size)

Calloc() requires 2 arguments count and size type

Count will provide no of elements

Size-type is data type size

EX:

Int \*arr;

arr=(int \*)calloc(10,size of(int));

char \*str;

str=(char\*)calloc(50,size of(char));

**Example:**

#include <stdio.h>

#include <stdlib.h>

int main()

{

int num, i, \*ptr, sum = 0;

printf("Enter number of elements: ");

scanf("%d", &num);

ptr = (int\*) calloc(num, sizeof(int));

if(ptr == NULL)

{

printf("Error! memory not allocated.");

exit(0);

}

printf("Enter elements of array: ");

for(i = 0; i < num; ++i)

{

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

sum += \*(ptr + i);

}

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

free(ptr);

return 0;

}

**realloc():**

changes memory size that is already allocated to a variable

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

by using realloc() we can create the memory dynamically at middle stage.

Realloc will creates the memory in bytes format and initial value is garbage.

Realloc will reqire 2 types of arguments of type void\* Size\_type()

**Example:**

#include <stdio.h>

#include <stdlib.h>

int main()

{

int \*ptr, i , n1, n2;

printf("Enter size of array: ");

scanf("%d", &n1);

ptr = (int\*) malloc(n1 \* sizeof(int));

printf("Address of previously allocated memory: ");

for(i = 0; i < n1; ++i)

printf("%u\t",ptr + i);

printf("\nEnter new size of array: ");

scanf("%d", &n2);

ptr = realloc(ptr, n2);

for(i = 0; i < n2; ++i)

printf("%u\t", ptr + i);

return 0;

}

**Free**:

When your program comes out operating system automatically release all the memory allocated by your program but as a good practice when you are not in a need of memory anymore then you should release that memory by calling the function free ().

Dynamically created malloc () or calloc() doesn’t get freed on its own you must explicitly use free() to release

Syntax:

Free(ptr);

**Example:**

#include <stdio.h>

#include <stdlib.h>

int main()

{

int num, i, \*ptr, sum = 0;

printf("Enter number of elements: ");

scanf("%d", &num);

ptr = (int\*) malloc(num \*sizeof(int)); //memory allocated using malloc

if(ptr == NULL)

{

printf("Error! memory not allocated.");

exit(0);

}

printf("Enter elements of array: ");

for(i = 0; i < num; ++i)

{

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

sum += \*(ptr + i);

}

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

free(ptr);

return 0;

}

**Files:**

When program is terminated entire data will be lost storing data in file will reserve your data even if the program terminates.

If we have large no. of data then we want type all that again takes a lot of time

However, if a file contains all the data, you can easily access the contents of the file using few commands in c.

We can easily move data from one computer to other.

That’s why we need file.

**File input and output functions:**

**Formatted functions**:

1. fscanf() for i/p function
2. fprintf() for o/p function

**unformatted:**

1. i/p fucntions like getc(), getw(), fread().
2. o/p functions like putc(), putw(), fwrite().

Files it is a collection of data that is stored on secondary devices like hard disk.

There are 2 types of files are there:

1. Text files
2. Binary files

**Text files:**

Text files are formed .txt files that you can easily create using notepad or any simple text editors

**Binary files:**

Binary files are mostly in .bin format

In text files we can easily edit or delete the contents.

Text files take minimum effort to maintain

It provides least security because everyone easily understood normal text.

They can hold high amount of data, are not readable easily and provide more security than text files because it is very difficult to understand.

Files operations:

1. Naming an exiting file
2. Opening data from file
3. Reading data from file
4. Writing data into a file
5. Closing a file

**Steps:**

Declaring a file

Declare a file pointer.

Open a file using the fopen() function.

Process the file using suitable file functions.

Close the file using the fclose() function.

**Declaration of file:**

When we are working with a file you need to declare a pointer type file

This declaration is needed to communicate between file and program

Syntax:

FILE \*fp;

Fp=fopen(“fileopen”, “mode”)

Fp=fopen(“e.programamin.txt”, “r”)

Fp=fopen(“e.programaming.txt”. “rb”)

**Closing a file**:

The file both text and binary should be closed whenever it is opened

Fclose()

Fclose(fptr);

r opens a text file in read mode

w opens a text file in write mode

a in append mode

r+ both write a d read operations ill done but first read operation will be done

w+ both write and read operation will be done but firstly write operation will be done after that read operation will be done

a+ both read and write operations will be done

rb pens a binary file in read mode

wb opens the binary file in write mode

ab opens a binary file in append mode

rb+ opens a binary file in read and write operations but read operation will be done firstly

wb+ opens a file in write and read operations but write operation will be done firstly.

Ab+ opens the file in binary form in both read and write operations

**What is the difference between write operation and append mode**

Both are used to open file in both files a new file is created when existing file is not open

**The only difference is that when we give append mode the cursor will move to the end of the program whereas in write mode the data that which is already there get erased and freshly it will open.**

**Formatted File I/O Functions**

fprintf (fp, “control string”, list);

**Example**:

fprintf(fp1, “%s %d”, name, age);

fscanf(fp, “control string”, list);

**Example:**

fscanf(fp, “%s %d”,& name, & age);

fscanf is used to read list of items from a file

fprintf is used to write a list of items to a file.

EOF – End of file (when EOF encountered the reading / writing should be terminated)

**Example1:**

#include main()

{

FILE \*fp;

fp = fopen("file.txt", "w");

//opening file

fprintf(fp, "Hello file by fprintf...\n");

//writing data into file

fclose(fp);//closing file

}

**Example2:**

#include <stdio.h>

int main()

{

int num;

FILE \*fptr;

fptr = fopen("C:\\program.txt","w");

if(fptr == NULL)

{

printf("Error!");

exit(1);

}

printf("Enter num: ");

scanf("%d",&num);

fprintf(fptr,"%d",num);

fclose(fptr);

return 0;

}

**Example 3**: Read from a text file using fscanf()

#include<stdio.h>

int main ()

{

int num;

FILE \*fptr;

if ((fptr = fopen("C:\\program.txt","r")) == NULL) {

printf("Error! opening file");

// Program exits if the file pointer returns NULL.

exit(1);

}

fscanf(fptr,"%d", &num);

printf("Value of n=%d", num);

fclose(fptr);

return 0;

}

Unformatted File I/O Functions

fputc() function

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

**putc ( ):-**Putting a character in to the file. It works with only character data type. One character at a time can write into a file.

Ex: char ch =‟a‟;

putc (ch, fp);

**Example:**

#include <stdio.h>

main(){

FILE \*fp;

fp = fopen("file1.txt", "w");//opening file

fputc('a',fp);//writing single character into file

fclose(fp);//closing file

}

**fgetc()** function

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

getc ( ): getting a character from the file, or reading the file information character by character at a time, upto the end of the file by using this function.

Ex: char ch; ch = getc (fp)

**Example:**

#include<stdio.h>

#include<conio.h>

void main(){

FILE \*fp;

char c;

clrscr();

fp=fopen("myfile.txt","r");

while((c=fgetc(fp))!=EOF){

printf("%c",c);

}

fclose(fp);

getch();

}

myfile.txt

**fputs()**

The fputs() function writes a line of characters into file

**Example:**

#include<stdio.h>

#include<conio.h>

void main(){

FILE \*fp;

clrscr();

fp=fopen("myfile2.txt","w");

fputs("hello c programming",fp);

fclose(fp);

getch();

}

**fgets()** The fgets() function reads a line of characters from file

**Example:**

#include<stdio.h>

#include<conio.h>

void main(){

FILE \*fp;

char text[300];

clrscr();

fp=fopen("myfile2.txt","r");

printf("%s",fgets(text,200,fp));

fclose(fp);

getch();

}

**getw and putw**

These are integer oriented functions. These are similar to above functions and are used to read and write integer values. These are useful when we deal with only integer data. The general format is

putw ( ): putting or writing of an integer value to a file.

putw (integer , fp);

Ex: int x = 5;

putw(x,fp);

getw ( ): getting or reading integer value from a file.

Ex: int x;

x = getw (fp);

**File Positioning Functions**

**fseek()** 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: fseek(FILE \* stream, long int offset, int whence)

The first parameter stream is the pointer to the file.

The second parameter is the position of the record to be found, and the third parameter specifies the location where the offset starts.

Different Whence in fseek Whence Meaning

SEKK\_SET Starts the offset from the beginning of the file.

SEKK\_END Starts the offset from the end of the file.

SEKK\_CUR Starts the offset from the current location of the cursor in the file. or

fseek(file pointer, offset, position);

file pointer is a pointer to the concerned file.

Offset is a number or variable of type long, it specifies the number of positions (bytes) to be moved from the location specified. If offset is positive number, then moving forward or negative meaning move backwards.

Position is a n integer number and it specifies from which position the file pointer to be moved. Position can take one of the following three values. 0 beginning of file 1 current position 2 end of file

Eg: fseek (fp, 0L,0); - go to the beginning of the file. (Similar to rewind).

fseek (fp, 0L,1); - Stay at current position (Rarely used)

fseek (fp, 0L,2); -go to the end of the file, past the last character of the file.

**Example:**

#include

void main()

{

FILE \*fp;

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

fputs("This is javatpoint", fp);

fseek( fp, 7, SEEK\_SET );

fputs("sonoo jaiswal", fp);

fclose(fp);

} myfile.txt

This is sonoo jaiswal

**rewind()** This function places the file pointer to the beginning of the file, irrespective of where it is present right now. It takes file pointer as an argument.

Syntax: rewind( fp);

**Example:**

#include

#include void main()

{

FILE \*fp;

char c;

clrscr();

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

while((c=fgetc(fp))!=EOF)

{

printf("%c",c);

}

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

while((c=fgetc(fp))!=EOF)

{

printf("%c",c);

}

fclose(fp);

getch();

}

Output: this is a simple text this is a simple text As you can see, rewind() function moves the file pointer at beginning of the file that is why "this is simple text" is printed 2 times. If you don't call rewind() function, "this is simple text" will be printed only once.

**ftell()** 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: n = ftell(fp);

n would give the relative offset(in bytes).

**Example:**

#include<stdio.h>

void main ()

{

FILE \*fp;

int length;

clrscr();

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

fseek(fp, 0, SEEK\_END); length = ftell(fp);

fclose(fp);

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

getch();

}

Output: Size of file: 21 bytes

**PRINTF FORMAT STRINGS**

%d : integer

%5d : integer in a field of width 5 chars

%-5d : integer in a field of width 5 chars, but adjusted to the left

%05d : integer in a field of width 5 chars, padded with zeroes from the left

%g : float variable in %f or %g notation

%e : float variable in scientific notation

%11.3e : float variable in scientific notation, with 3 decimals, field of width 11 chars

%5.1f : float variable in fixed decimal notation, with one decimal, field of width 5 chars

%.3f : float variable in fixed decimal form, with three decimals, field of min. width

%s : string

%-20s : string in a field of width 20 chars, and adjusted to the left

**Typedef:**

The typedef is a keyword that allows the programmer to create a new data type name for the exisiting data type.so,the purpose of the typedef is to redefine the name of an existing variable type.

Typedef datatype alias\_name;

**Bitfields:**

The variables defined with predefined width is called as bit fields.

The no.of bits in bitfield ,the width must be less than or equal to the width of specified data type.

Need of Bitfield:

Reduces memory consumption

**Errors**

There are mainly 5types of errors:

* Syntax Error
* Run Time Error
* Linker Error
* Logical Error
* Semantic Error

**Syntax Errors:**

This kind of errors are generally indicated by the compiler before compilation.

Also known as compile time error:

easily corrected or debugged.

Most frequency syntax errors are:

Missing parenthesis (})

Printing the value without declaring it

Missing semicolon

int a;

Int a;//incorrect form

These errors generally occurred due to mistakes while typing or do not follow the syntax of the specified language.

**Run time errors:**

When the program is running and it is not able to perform operation is the main cause for it.

Errors which are occur during program execution after compilation are called

Run time errors.

One of the most common errors is the division by zero also known as ero division error.

Compilation does not find these errors.

#include<stdio.h>

int main()

{

Int a=2;

Int b=2/0;

Printf(“the value b is %d”,b);

return 0;

}

**Linker error:**

Linker error is mainly generated when the executable file of the program is not created.

**Logical error:**

The logical error is error that leads to an undesired output.

These errors produce the incorrect output, but they are error free, known as logical errors.

These types of mistakes are mainly done by beginners.

These errors produce the incorrect output.

For(i=0;i<=10;i++);

**Semantic error:**

When the statements are not understandable by the complier these types of error occur.

#include<stdio.h>

Int main()

{

Int a,b,c;

a=2;

b=3;

c=1;

a+b=c;

return 0;

}

In c there are two types of i/p and o/p functions are available in c

1.Formatted

2.Unformatted

**Formatted:**

Here it represents all data type

**Unformatted**

Here in unformatted only character and string.

**Formatted:** They are in a formatted order that which we have to follow.

**printf():** output data are result of the operation is displayed

**scanf():** input data can be entered into the computer

**unformatted**: which don’t have follow any order.

1. character i/o functions
2. string i/o function

**Character:**

1. getchar(): Used to read a character from the standard input

2. putchar(): Used to display a character to standard output

3. getch() and getche(): these are used to take the any alpha numeric characters from the standard input getche() read and display the character getch() only read the single character but not display

4.putch(): Used to display any alpha numeric characters to standard output

**String I/O:**

1. gets(): Used for accepting any string from the standard input(stdin) eg:gets()

2. puts(): Used to display a string or character array Eg:puts()

3. Cgets():read a string from the console eg; cgets(char \*st)

4. Cputs():display the string to the console eg; cputs(char \*st)

**getchar() function** : A single character can be given to the computer using „C" input library

function getchar().

Syntax : char variable=getchar();

The getchar() function is written in standard I/O library. It reads a single character from a

standard input device. This function does not require any arguments, through a pair of

parentheses, must follow the statements getchar().

#include<stdio.h>

#include<conio.h>

#include<ctype.h>

void main()

{

char ch;

clrscr();

printf("Enter any character/digit:");

ch=getchar();

if(isalpha(ch)>0)

printf("it is a alphabet:%c\n",ch);

else if(isdigit(ch)>0)

printf("it is a digit:%c\n",ch);

else

printf("it is a alphanumeric:%c\n",ch);

getch();

}.

OUTPUT: Enter any character/Digit : abc

it is a alphabet:a

2 : putchar() function :The putchar() function is used to display one character at a time on the standared output device. This function does the reverse operation of the single character input

function.

Syntax : putchar(character varaiable);

#include<stdio.h>

#include<conio.h>

#include<ctype.h>

void main()

{

char ch;

printf("Enter any alphabet either in lower or uppercase:");

ch=getchar();

if(islower(ch))

putchar(toupper(ch));

else

putchar(tolower(ch));

getch();

}

OUTPUT: Enter any alphabet either in lower or uppercase: a

A

**gets() :** The gets() function is used to read the string (String is a group of characters) from the standard input device (keyboard).

Syntax : gets(char type of array variable);

Ex :#include<stdio.h>

#include<conio.h>

void main()

{

char str[40];

clrscr();

printf("Enter String name:");

gets(str);

printf("Print the string name%s:",str);

getch();

}

OUTPUT : Enter the string : reddy

Print the string :reddy

**puts() :**The puts() function is used to display the string to the standard output device

(Monitor).

Syntax : puts(char type of array variable);

Program using gets() function and puts() function.

#include<stdio.h>

#include<conio.h>

void main()

{

char str[40];

puts("Enter String name:");

gets(str);

puts("Print the string name:");

puts(str);

getch();

}

OUTPUT :Enter string name :

subbareddy

Print the string name

subbareddy

**getch() function** :The getch function reads a single character directly from the keyboard, without echoing to the screen.(for password purpose we will use getch() function.getch() means here it will be like that until we enter any character if we don’t give getch then it will display only at once after it will automatically close.

Syntax :

int getch();

Ex : #include<stdio.h>

void main()

{

char c;

c=getch();

}

**getche() function** :The getche() function reads a single character from the keyboard and echoes it to the current text window.(echoes means shows or dispalys it on screen)

Syntax : int getche();

Ex : #include<stdio.h>

void main()

{

char c;

c=getche();

}

**getc() function :** This function is used to accept a single character from the standard input to a

character variable.

Syntax : character variable=getc();

**putc() function** :This function is used to display a single character in a character variable to

standard output device.

Syntax : putc(character variable);

**Precedence and associativity:**

.

/\*#include<stdio.h>

int main()

{

int a=3,b=4,c=8;

printf("%d",a);

return 0;

}\*/

/\*#include<stdio.h>

int main()

{

int a;

a=3,4,8;

printf("%d",a);

return 0;//output 3

comma operator is having least precedence among all the operators available in c language.

**Some important points to remember:**

Expressions have also classified that the type that they represent:

Boolean expression

Integer expression

Floating point

String expression

#include<stdio.h> is preprocessor complier directive it is not the statement so is does not end with semicolon

**We know that computer only understands 0’s and 1’s when we give a variable how can it convert to machine language**.

Bu using ASCII American Standard code for Information Interchange

**Why we are using return 0?**

The return 0 value tells us that the program has completed all its task and has terminated successfully and the hardware resources that has been allocated to the program can be now freed and it will be reallocated to other program. Here we can put any number in place of zero either positive or negative it will send the message that the program has been terminated but only zero will send a successful and error free termination message.

**Round a number**:

Ceil():function rounds a number upwards to its nearest integer.

Floor():function rounds a number downwards to its nearest integer.

**Example:**

#include<stdio.h>

#include<math.h>

int main()

{

float a=3.4;

printf("%f",ceil(a));//4.0000

printf("%f",floor(a));//3.000

return 0;

}

**Twist in bitwise complement operator in C Programming**

The bitwise complement of 35 (~35) is -36 instead of 220, but why? For any integer n, bitwise complement of n will be -(n+1).

**Infinitive loop:**

Infinitive for loop in C If you don't initialize any variable, check condition and increment or decrement variable in for loop, it is known as infinitive for loop. In other words, if you place 2 semicolons in for loop, it is known as infinitive for loop for(; ;).

Compiler and interpreter both task is same but the process id different Compler takes all the code at a time and run it’ where as interpreter takes the line by line and execute it.

For large programs we preferred to use complier.

**Recursion** When function is called within the same function, it is known as recursion in C.

A function that calls itself, and doesn't perform any task after function call, is know as tail recursion. In tail recursion, we generally call the same function with return statement.

**Features :**

• There should be at least one if statement used to terminate recursion.

• It does not contain any looping statements.

**Advantages:**

• It is easy to use.

• It represents compact programming structures.

**Disadvantages:**

• It is slower than that of looping statements because each time function is called.

Note: while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop. Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

Example of recursion.

recursionfunction()

{

recursionfunction();//calling self function

}

**Example of tail recursion in C // print factorial number using tail recursion**

#

int factorial (int n)

{

if ( n < 0)

return -1; /\*Wrong value\*/

if (n == 0)

return 1; /\*Terminating condition\*/

return (n \* factorial (n -1));

}

void main()

{

int fact=0;

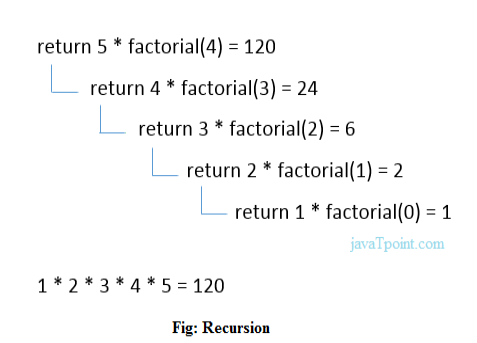
fact=factorial(5);

printf("\n factorial of 5 is %d",fact);

return 0;

}

Outputfactorial of 5 is 120



**list of preprocessor directives.**

* #include
* #define
* #undef
* #ifdef
* #ifndef
* #if
* #else
* #elif
* #endif
* #error
* #pragma

**C Macros**

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

1. Object-like Macros

2. Function-like Macros

**Object-like Macros**

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

For example:

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

**Function-like Macros**

The function-like macro looks like function call.

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

**For example**:

#include stdio.h

main()

{

printf("File :%s\n", \_\_FILE\_\_ );

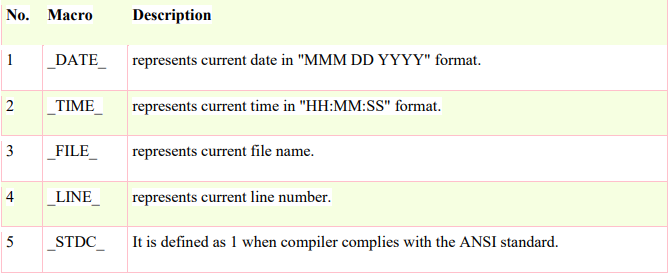
printf("Date :%s\n", \_\_DATE\_\_ );

printf("Time :%s\n", \_\_TIME\_\_ );

printf("Line :%d\n", \_\_LINE\_\_ );

printf("ANSI :%d\n", \_\_STDC\_\_ );

}



**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

2. #include "filename"

The #include 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 notes:

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

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

C #define

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

Syntax:

#define token value

**Example**

#define to define a constant.

#include stdio.h>

#define PI 3.14

main()

{

printf("%f",PI);

}

Output: 3.140000

**C #undef**

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

**Syntax:**

#undef token

Simple example to define and undefine a constant.

#include stdio.h>

#define PI 3.14

#undef PI main()

{

printf("%f",PI);

}

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.

#include<stdio.h>

#define number 15

int square=number\*number;

#undef number

main()

{

printf("%d",square);

}

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:

#ifdef MACRO //code

#endif

Syntax with

#else:

#ifdef MACRO //successful code

#else //else code

#endif C

#ifdef example

#include

#include

#define NOINPUT

void main()

{

int a=0;

#ifdef NOINPUT

a=2;

#else printf("Enter a:");

scanf("%d", &a);

#endiprintf("Value of a: %d\n", a);

getch();

}

Output: Value of a: 2

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

#include

#include

void main()

{

int a=0;

#ifdef NOINPUT

a=2;

#else

printf("Enter a:");

scanf("%d", &a);

#endif

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

getch();

}

Output: Enter a:5 Value of a:

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:

#ifndef MACRO //code

#endif

Syntax with #else:

#ifndef MACRO //successful code

#else //else code

#endif C

#ifndef preprocessor directive.

#include

#include

#define INPUT

void main()

{

int a=0;

#ifndef INPUT a=2;

#else printf("Enter a:");

scanf("%d", &a);

#endif

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

getch();

}

Output: Enter a:5

Value of a: 5

if you don't define INPUT, it will execute the code of #ifndef.

#include

#include

void main()

{

int a=0;

#ifndef INPUT

a=2;

#else

printf("Enter a:");

scanf("%d", &a);

#endif

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

getch();

}

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:

#if expression //code

#endif

Syntax with #else:

#if expression //if code

#else //else code

#endif

Syntax with #elif and #else:

#if expression //if code

#elif expression //elif code

#else //else code

#endif

C #if example

#include

#include

#define NUMBER 0

void main()

{

#if (NUMBER==0)

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

#endif

getch();

}

Output: Value of Number is: 0

Another example to understand the #if directive clearly.

#include

#include

#define NUMBER 1

void main()

{

clrscr();

#if (NUMBER==0)

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

#endif

#if (NUMBER==1)

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

#endif

getch();

}

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:

#if expression //if code

#else //else code

#endif

Syntax with #elif:

#if expression //if code

#elif expression //elif code

#else //else code

#endif C

#else example

#include

#include

#define NUMBER 1

void main()

{

#if NUMBER==0

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

#else

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

#endif

getch();

}

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

#include

#ifndef \_\_MATH\_H

#error First include then compile

#else

void main()

{

float a;

a=sqrt(7);

printf("%f",a);

}

#endif

Output: Compile Time Error

First include then compile if you include math.h, it does not gives error. #include

#include

#ifndef \_\_MATH\_H

#error First include then compile

#else

void main()

{

float a;

a=sqrt(7);

printf("%f",a);

}

#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: #pragma token Different compilers can provide different usage of #pragma directive.

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

#pragma argsused

#pragma exit

#pragma hdrfile

#pragma hdrstop

#pragma inline

#pragma option

#pragma saveregs

#pragma startup

#pragma warn

Example to use #pragma preprocessor directive.

#include

#include

void func() ;

#pragma startup func

#pragma exit func

void main()

{

printf("\nI am in main");

getch();

}

void func()

{

printf("\nI am in func");

getch();

}

Output: I am in func I am in main I am in func

**Key Points To Remember**:

1. Source program is converted into executable code through different processes like precompilation, compilation, assembling and linking.

2. Local variables uses stack memory.

3. Dynamic memory allocation functions use the heap memory.

preprocessor Syntax/Description Macro Syntax:

#define This macro defines constant value and can be any of the basic data types. Header file inclusion

Syntax: #include The source code of the file “file\_name” is included in the main program at the specified place. Conditional compilation

Syntax:

#ifdef,

#endif,

#if,

#else,

#ifndef

Set of commands are included or excluded in source program before compilation with respect to the condition. Other directives

Syntax: #undef,

#pragma

#undef is used to undefine a defined macro variable.

#Pragma is used to call a function before and after main function in a C program.

There are three types of preprocessor commands.

1 : macro substitution.

2 : file inclusion.

3 : conditional compilation directives.

1: Macro Substitution : They are two types of macro substitution.

1 : Macro substitution without arguments.

2 : Macro substitution with arguments.

1 : Macro substitution without arguments :

It is a process to substitute the constant or value in the place of an identifier. It is possible to achieve this with the help of directive or macro definition statement

#define.

Syntax :

#define identifier constant or expression

Ex :

#define PI 3.142

#define MAX\_MARKS 100

#define MIN\_MARKS 35

Ex :

#include

#define height 100

#define number 3.14

#define letter 'A'

#define letter\_sequence "ABC"

#define backslash\_char '\?'

void main()

{

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

printf("value of number : %f \n", number );

printf("value of letter : %c \n", letter );

printf("value of letter\_sequence : %s \n", letter\_sequence);

printf("value of backslash\_char : %c \n", backslash\_char);

}

OUTPUT:

value of height : 100

value of number : 3.140000

value of letter : A

value of letter\_sequence : ABC

value of backslash\_char : ?

Ex : Example of Macro substitution

#include

#define PI 3.142

void main()

{

int r;

float area;

printf(“Enter the radius of circle”);

scanf(“%d”,&r); area=PI\*r\*r;

printf(“the area of a circle is%d”,area);

}

Example of Macro definition with expressions

#define A (20\*10)

#define B (200-100)

void main()

{

int div;

div=A/B;

printf(“the division of two numbers%d”,div);

}

Example of Macro definition with conditional expression

#define

IFCONDITION

if(a>b)

#define PRINT

printf(“the value of a is the greatest no”)

void main()

{

int a=100,b=50;

IFCONDITION PRINT;

}

Macro Substitution with Arguments :

Syntax :

#define identifier(var1,var2,va3,….varn)

string Where identifier is the name of macro function with the list of macro formal parameters var1,var2,var3,…varn like the formal parameters in a function definition.

Ex :

#define PROD(x) (x\*x)

void main()

{

int a,mul;

printf(“enter the value of a”);

scanf(“%d”,&a);

mul=PROD(a);

printf(“The multification of two numbers%d”,mul);

}

2 : FILE INCLUSION : A copying of one file to another files into program. Ex : File inclusion of an external file “add.c”.

#include

#include

add.c

void main()

{

void add(); //FUNCTION PROTOTYPE/ DECLARATION.

add(); //FUNCTION CALLING

}

The file add1.c contains the function definition as follows.

void add()

{

int a,b,c;

printf(“enter two numbers”);

scanf(%d%d”,&a,&b);

c=a+b;

printf(“c value is:%d”,c);

}

3 : CONDITIONAL COMPILATION DIRECTIVES : C preprocessor also supports number of conditional compilation directives as

1 :#undef : Undefined a macro

2 : #ifdef : Tests for a macro definition.

3 : #endif : Specifies the end of #if.

4 : #if : Tests compile-time condition.

5 : #else : Specifies alternative when #if test fails.

These are used to select a particular segment of code for compilation depending on the condition.

EXAMPLE PROGRAM FOR CONDITIONAL COMPILATION DIRECTIVES: #IFDEF, #ELSE AND #ENDIF IN C:

• “#ifdef” directive checks whether particular macro is defined or not. If it is defined, “If” clause statements are included in source file.

• Otherwise, “else” clause statements are included in source file for compilation and execution.

Ex:

#include

#define RAJU 100

int main()

{

#ifdef RAJU

printf("RAJU is defined. So, this line will be added in " \ "this C file\n");

#else printf("RAJU is not defined\n");

#endif

return 0;

}

OUTPUT: RAJU is defined. So, this line will be added in this C file

B) EXAMPLE PROGRAM FOR #IFNDEF AND #ENDIF IN C:

• #ifndef exactly acts as reverse as #ifdef directive. If particular macro is not defined, “If” clause statements are included in source file.

• Otherwise, else clause statements are included in source file for compilation and execution.

Ex:

#include

#define RAJU 100

int main()

{

#ifndef SELVA

{

printf("SELVA is not defined. So, now we are going to " \ "define here\n"); #define SELVA 300

}

#else printf("SELVA is already defined in the program”);

#endif

return 0;

}

OUTPUT: SELVA is not defined. So, now we are going to define here C) EXAMPLE PROGRAM FOR #IF, #ELSE AND #ENDIF IN C:

• “If” clause statement is included in source file if given condition is true.

• Otherwise, else clause statement is included in source file for compilation and execution.

Ex:

#include

#define a 100

int main()

{

#if (a==100)

printf("This line will be added in this C file since " \ "a \= 100\n");

#else

printf("This line will be added in this C file since " \ "a is not equal to 100\n"); #endif

return 0;

}

OUTPUT: This line will be added in this C file since a = 100 EXAMPLE PROGRAM FOR UNDEF IN C LANGUAGE: This directive undefines existing macro in the program.

Ex:

#include

#define height 100

void main()

{

printf("First defined value for height : %d\n",height);

#undef height // undefining variable

#define height 600 // redefining the same for new value

printf("value of height after undef \& redefine:%d",height);

}

OUTPUT: First defined value for height : 100 value of height after undef & redefine : 600

EXAMPLE PROGRAM FOR PRAGMA IN C LANGUAGE: Pragma is used to call a function before and after main function in a C program.

Ex:

#include

void function1( );

void function2( );

#pragma startup function1 #pragma exit function

int main( )

{

printf ( "\n Now we are in main function" ) ;

return 0;

}

void function1( )

{

printf("\nFunction1 is called before main function call");

}

void function2( )

{

printf ( "\nFunction2 is called just before end of " \ "main function" ) ;"

}

OUTPUT: Function1 is called before main function call Now we are in main function Function2 is called just before end of main function

Ex :

#define TEST 1

void main()

{

#ifdef TEST

{

printf(“This is compiled”);

}

#else

{

printf(“This is not compiled”);

}

#endif

}

Example:

#define FLAG 1

char ch;

void main()

{

#if FLAG

{

ch=‟t‟;

printf(“This is compiled”);

}

#else

{

ch=‟f‟;

printf(“This is not compiled”);

}

#endif

}

**STANDARD C LIBRARY FUNCTIONS**

1 : stdio.h

2 : stdlib.h

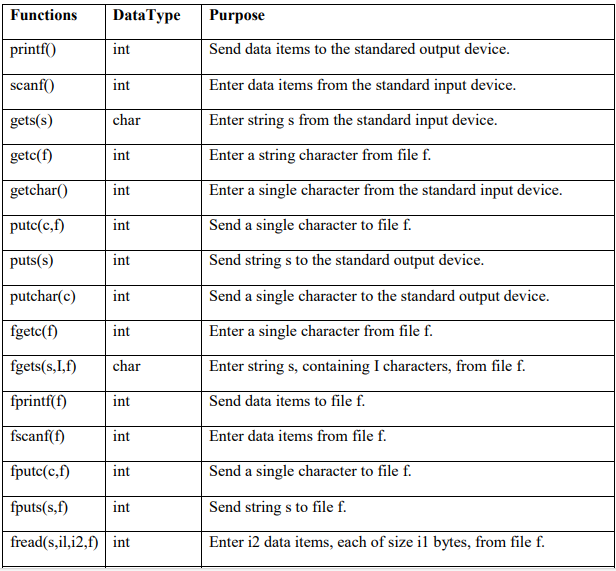
3 : string.h

4 : math.h

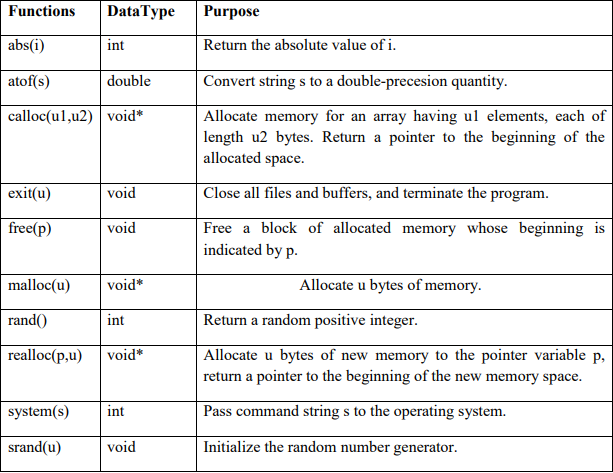
5 : ctype.h

6 : time.h

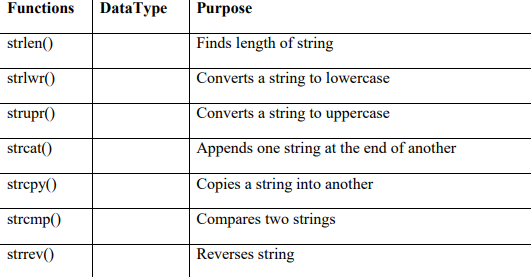
**Stdio.h**

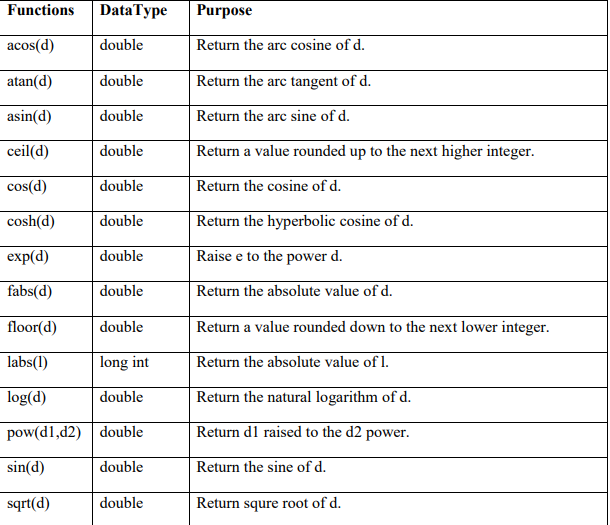
****

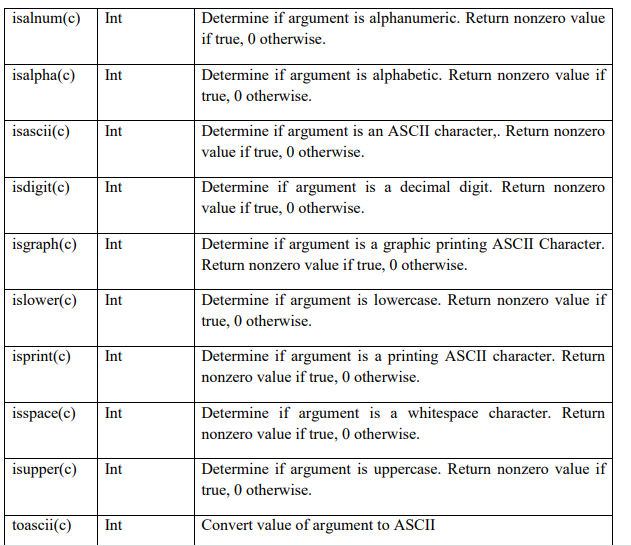
**Stdlib.h**

****

**String:**

****

****

****