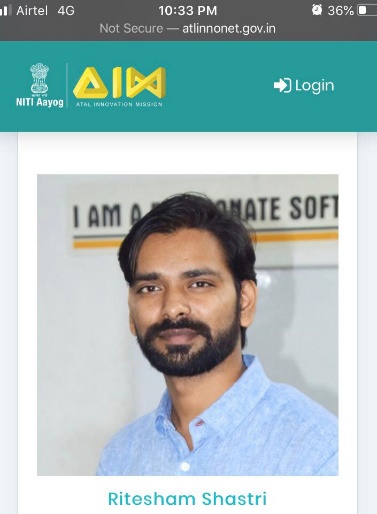
THE

C

PROGRAMMING

LANGUAGE

**Dennis Ritchie**

 ' 

**By**:-Deepak Verma **MENTOR** :-RITESHAM SHASTRI

Note:- C is a relatively low-level programming language because c is directly interact with the hardware.

# **Why C language is so important?**

* C is the most popular programming language in whole world.
* C cover basic feature of all programming language.
* C is the most popular language for hardware Independent programming.
* Oracle is written in C.
* Core libraries of android are written in C.
* Unix operating system is developed in C.
* Almost every device driver is written in C.

# **Features of C language:-**

* General purpose Programming language
* Structured programming language
* Portable/Machine Independent
* Modular Programming Language
* Case-Sensitive language
* Rich Library
* Middle-level programming language
  + Direct Access to memory through pointer
  + Bit manipulation using bitwise operator
  + Writing assembly code within c code
* Easy to Understand

# **History of C Language**

**John Backus: -**

* Developer of FORTRAN(Formulae Translation)
* 1957
* Used for mathematical calculation

**CODASYL :-**

* Developer of COBOL(Commercial Business Oriented Language)
* Around 1959
* Used for business software

**Martin Richard:-**

* Developer of BCPL(basic combined programming language )
* Around 1966
* Used for general purpose

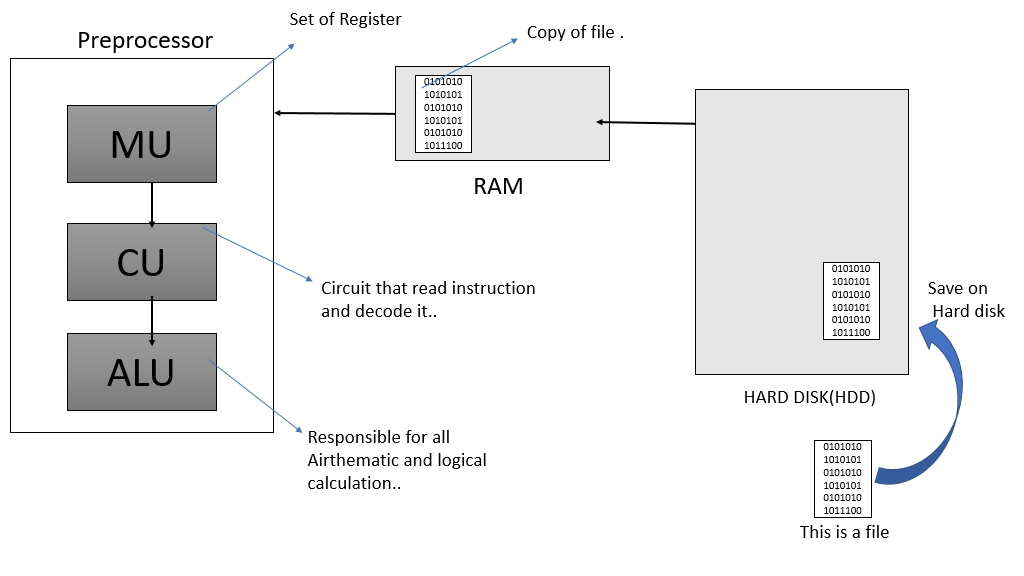
**Ken Thompson:-**

* Developer of B language.
* Around 1969
* Also Developer of Unix Operating System.

**Dennis Ritchie: -**

* Developer of C language
* 1972
* at AT&T’s Bell LABs, USA.
* Co-developer of Unix OS.

# **How to execute a programme:-**



# **PROGRAMMING**

* Programming is a process of taking an **Algorithm** and Coding it into a notation.
* Example:-C, C++, JAVA , PHP , Perl , Python ,SQL etc.

Note:-**Algorithm** is a optimize solution to a particular problem.

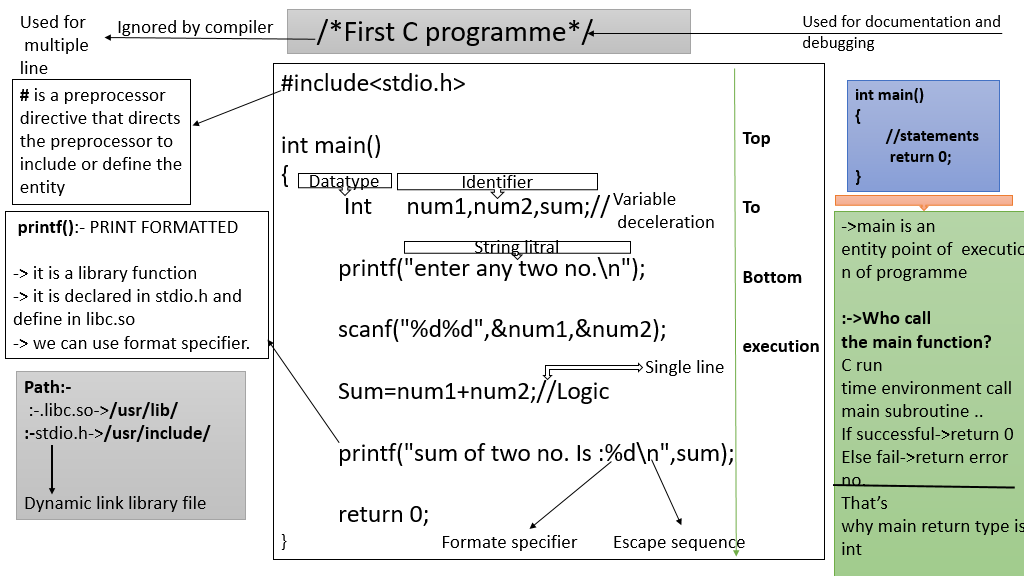
## Types of Programming language: -

* Low-level -> Assembly
* Procedure Oriented-> C, pascal.
* Object Oriented-> C++ , Java , C#
* Programming for Web-> HTML , PHP.
* Programming for database-> Sql
* A Scripting-> Shell, JavaScript

# **Structured of C language:-**

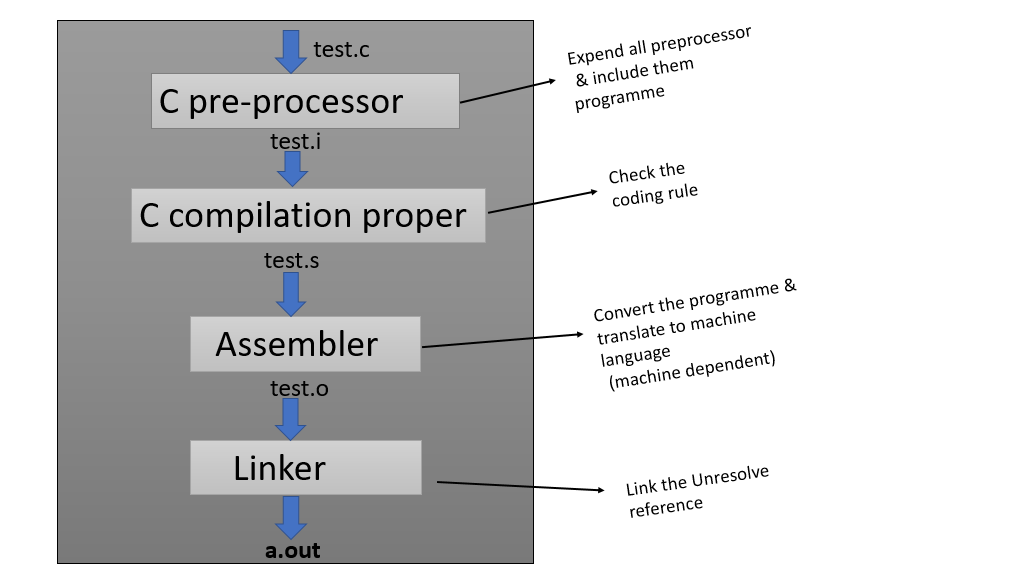
1. **DOCUMENTATION**
2. **LINK SECTION**
3. **DEFINATION SECTION**
4. **GLOBAL DECLERATION**
5. **MAIN FUNCTION**
6. **LOCAL DECLERATION**
7. **SUB-PROGRAME SECTION**

:----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------:



# **:-> C programme compilation steps <-:**

**GCC (GNU C compiler)**



NOTE:-command to see iso file ?

**:-> gcc -save-temp/temps (file name).c**

Identifier:---

**Constant:-**

* Any information is a constant
* Data=information=Constant.

**Variable:-**

* Variable is a name of memory allocation where we store a data.
* Variable can be a combination of alphabate,alphanumeric and underscores...
* A valid Variable name can not start with digit.

**Rules:-**

* Variable can Capitals and small alphabeted
* Number should not come first
* Space and special character not allowed
* Allowed only underscore

**Keywords:--**

* Predefine words
* Reserved words
* There are 32 Keywords in C language acc. To 16 bit architecture

**Ex**:- main() ,calloc(),malloc,sizeof,for,while etc.

**Some Important special character name:---**

* <> :- Angular bracket
* {} :- braces
* ( ) :- bracket or parenthis
* “is”:- Double Quote
* ‘a’ :- Single Quote
* % :- Modulus
* & :- Empersand
* \* :- Astrix or pointer or multiple
* [] :- Square Bracket
* / :- Forward slash
* \:- Backword slash

**Escape sequence: -**

* + \n :- used for next line
  + \t :- used for tab
  + \b:- used for backspace
  + \\:- used to print slash \
  + \”:- used to print double quote
  + \’:- used to print single quote
  + \a:-used to produce alert or beep sound
  + %% :- used to print %

**Format Specifier:-**

* %d , %i:- int
* %c:-char
* %f , %e ,%E:-float
* %g or %G:-float
* %lf:-double
* %u:-unsigned
* %x:-pointer(rarely) and hexadecimal

**Header File: -**

* Header file is a file have extension is .h
* **it contains: -**
  + pre-processor declarations
  + Standard variable declaration
  + Constant declaration
  + Function declaration
  + Macro
* **It is a two type: -**
  + - User define header file
    - Pre-define header file
* **Written type: -**
  + - **#include”stdio.h”**
    - **#include<stdio.h>**

**Note: -**

* + **Angular bracket** Header file **is a By default set location to explain Compiler..**

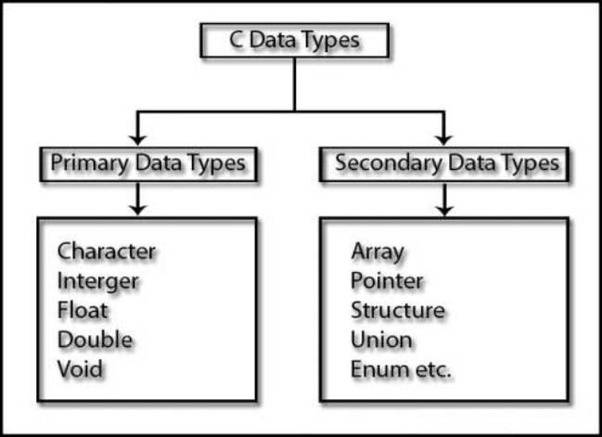
**Ex:- \usr\include\**

* + **In “ “ header file means to set location of header file that’s Understand Compiler**

**Ex:- “C:\user\file\stdio.h”**

# **Data type in ‘C’: -**

* C language has extensive range of data types which we can use programme. These datatype have different size and capacity.



* Variables **in C** are associated with **data type**. Each **data type** requires an amount of memory and performs specific operations. There are some common **data types in C** …

## **Integer Types**

The following table provides the details of standard integer types with their storage sizes and value ranges −

|  |  |  |
| --- | --- | --- |
| **Type** | **Storage size** | **Value range** |
| Char | 1 byte | -128 to 127 or 0 to 255 |
| unsigned char | 1 byte | 0 to 255 |
| signed char | 1 byte | -128 to 127 |
| Int | 2 or 4 bytes | -32,768 to 32,767 or -2,147,483,648 to 2,147,483,647 |
| unsigned int | 2 or 4 bytes | 0 to 65,535 or 0 to 4,294,967,295 |
| Short | 2 bytes | -32,768 to 32,767 |
| unsigned short | 2 bytes | 0 to 65,535 |
| Long | 8 bytes or (4bytes for 32 bit OS) | -9223372036854775808 to 9223372036854775807 |
| unsigned long | 8 bytes | 0 to 18446744073709551615 |

Note: - **To get the exact size of a type or a variable on a particular platform, you can use the sizeof operator. The expressions *sizeof(type)* yields the storage size of the object or type in bytes.**

## **Floating-Point Types**

The following table provide the details of standard floating-point types with storage sizes and value ranges and their precision −

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Storage size** | **Value range** | **Precision** |
| Float | 4 byte | 1.2E-38 to 3.4E+38 | 6 decimal places |
| Double | 8 byte | 2.3E-308 to 1.7E+308 | 15 decimal places |
| long double | 10 byte | 3.4E-4932 to 1.1E+4932 | 19 decimal places |

**Note**:**-printf() function by default give precision 6 decimal**

**Declaration: -**

* Declaration just explain the entity to compiler
  + EX: - int a; //a is a type of int;
  + float num1; // num1 is a type of float

**Initialization: -**

* At the time of declaration assigning value to it......

EX: -int a=10, b=20;

**Definition: -**

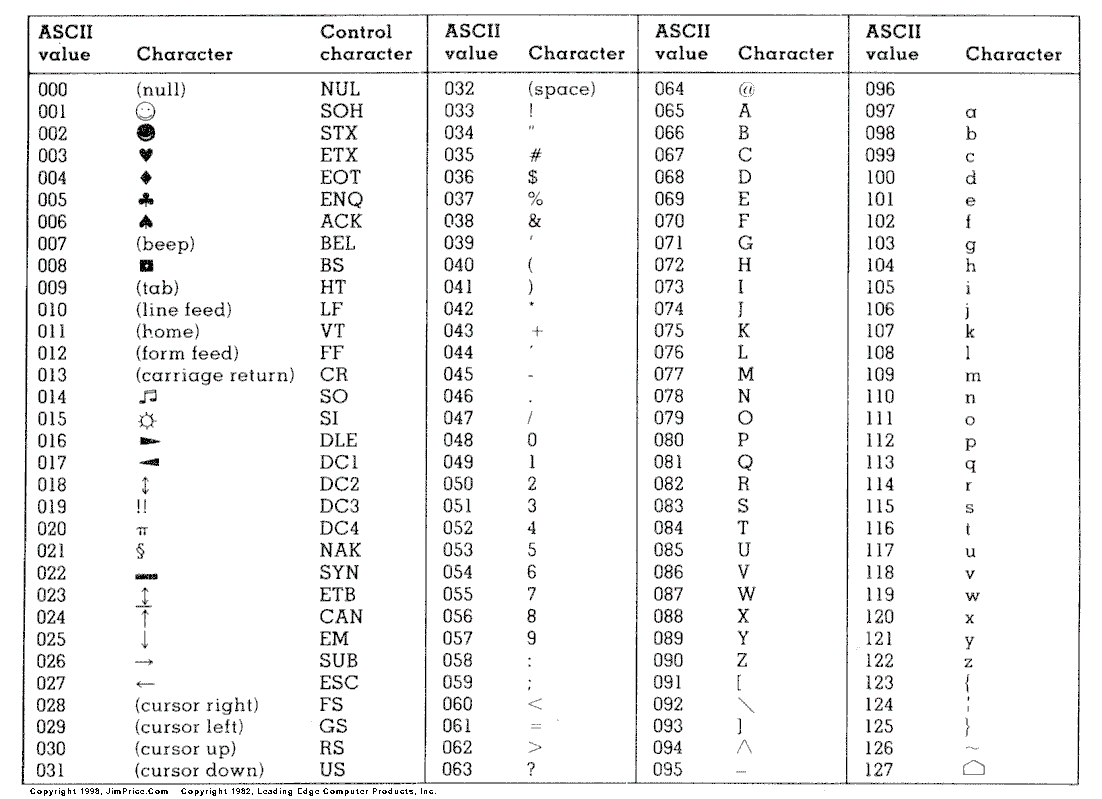
* When memory Associated after declaration.

EX: -int A; // define

A=10; // define

# **A S C I I Table**

* **ASCII**  stands for American Standard Code for Information Interchange.
* There are 128 Standard ASCII characters, numbered from 0 to 127. Extended ASCII adds another 128 value and give to 255



Note:- **‘A’+32=’a’ , ‘z’-32=’Z’**

**Note:-** space :- 32

Note: - **gotoxy(\_,\_) -> Function shift cursor every position...**

**Ex:-**

#include<stdio.h>

void gotoxy(int,int);

Int main()

{

clrscr(); // to clear the screen

gotoxy(40,13); // 40,13 is a co-ordinates of x and y

printf(“deepak\n”);

return 0;

}

**void** gotoxy(**int** x,**int** y)

{

**printf**("%c[%d;%df",0x1B,y,x);

}

**Note:-**

* + printf() is not a keyword
  + printf() is a pre-define function

**Note:-**

* + Unary operator those operators which is used in single operands. Ex: - x++;
  + Binary operators those operators which is used two operands Ex :- x+y; //here x,y is a operand
  + Ternary operator those operators which is used in triple operands Ex: - **z?x:y;**

**Note:-**

* + **sizeof()** operator used only single operands and it give size of data type of operands
  + **sizeof()** operator is a compile time Operator.

# **Arithmetic Instruction in C Language**

* Operator: -
  + - * 3+4 //here 3 and 4 is operands and + is operator

## Arithmetic instruction: -

* + - * An instruction which is used to manipulate data by using operator, is known as Arithmetic Instruction.

## Types of Operator: -

* + - * Unary operator-> **+, -, ++, --,sizeof()**
      * Arithmetic operator ->**+ ,- ,\*,/,%**
      * Relation operator-> **<,<=,>,>=,==,!=**
      * Logical operator->**&& , || , !**
      * Bit-wise operator-> **& , | , ~ , << , >> , ^**
      * Conditional(ternary) operator-> **? :**
      * Assignment operator-> **+=,-=,\*=,/=,%=** (compound operator)

**Note:-**

* + - **++ and -- are known as iteration**
    - **There are two iteration.**

**Pre-increment-> ++x**

**->First inc. And then assign**

**Post-increment-> x++**

**->First asign and then inc.**

**Note:- this concept basically used in printf()**

## Precedence & Associativity: -

* + - * This rule of priority of operators is called **operator precedence**.
      * Associativity is only used when there are two or more operators of same precedence.

|  |  |  |
| --- | --- | --- |
| **Category** | **Operator** | **Associativity** |
| Postfix | () [] -> . ++ - - | Left to right |
| Unary | + - ! ~ ++ - - (type)\* & sizeof | Right to left |
| Multiplicative | \* / % | Left to right |
| Additive | + - | Left to right |
| Shift | << >> | Left to right |
| Relational | < <= > >= | Left to right |
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | | | Left to right |
| Logical AND | && | Left to right |
| Logical OR | || | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= \*= /= %=>>= <<= &= ^= |= | Right to left |
| Comma | , | Left to right |

**Note:-**

* + - Real Constant consume memory of 8 bytes **like:- 77.18**
    - %(modulus) operator only work on integer value
    - Character in sizeof operator convert into ASCII value which means integer and occupied 4 bytes of memory **EX:- sizeof(‘a’)**

**Note: -**

* + - Int/int = int **Ex: - 3/2 = 1**
    - Int/float = float **Ex: - 3/2.0 = 1.0**
    - Float/int = float **Ex: - 3.0/2 = 1.0**
    - Float /float =float **Ex: - 3.0/2.0 = 1.0**

**Note: -**

**Modulus operator give the sign of numerator always**

* + - **4%3 = 1**
    - **-4%3 = -1**
    - **4%-3 = 1**
    - **-4%-3 = - 1**

# **Decision making/Conditional statement/Selection statement**

* I**f statement** (single selection statement)
* **If-else statement** (double selection statement)
* **If elseif...elseif..........else** (multiple selection)
* **Switch (choice/case based** selection)
* **Conditional/Ternary** ( ? : )

## If statement

#include<stdio.h>

Int main()

{

Int x;

printf(“enter a no\n”);

scanf(“%d”,&x);

If(x>0)

printf(“no. Is positive”);

return 0;

}

Note:-

* + - * don’t put semicolon after if()
      * We can remove braces in if() when in block single statement

## If-else statement

#include<stdio.h>

Int main()

{

Int age;

printf(“enter your age\n”);

scanf(“%d”,&age);

If(age>=18)

printf(“you are valid to play game”);

Else

printf(“you are not a valid to play “);

return 0;

## If-elseif statement

#include<stdio.h>

Int main()

{

float p ;

printf(“enter a percentage of marks\n:”);

scanf(“%f”,&p);

If (p>0 && p<33)

{

printf(“fail\n”);

}

elseif (p>=33 && p<45)

{

printf(“3rd division\n”);

}

elseif(p>=45&&p<60)

{

printf(“2nd division\n”);

}

elseif(p>=60)

{

printf(“1st division\n”);

}

else

{

printf(“ invalid percentage input\n”);

}

return 0;

**}**

## Nested if-else

**Syntax:**

If (condition)

{

If (condition)

{

//statement;

}

}

else

{

//statements;

}

**Note: -**

* + If float a=3.4 it means a assign value is 3.3999999999..... that means if assing float value is bit decrease.

**Note:-** use of **goto** keyword

**Syntax:**

goto there; // there is a lable..

…...............

................

…...........................

there:

printf(“deepak\n”);

## Switch

**Syntax:**

switch (expression)

{

case constant:

//statement;

break;

case constant:

//statement;

break;

case constant:

//statement;

break;

case constant:

//statement;

break;

case constant:

//statement;

break;

default:

//statement;

}

**Note:-**

* + - constant is anything just like a digit,alphabate **etc.**
    - The keyword **break** can be used in loop body or in switch body.
    - The purpose of **break** is to terminate loop execution immediately as it encounters.

# **Iterative / Repeatative Control Instruction(LOOPs):----------------**

* **Entry control loops:**
* While loop
* For loop
* **Exit control loops:**
* Do while loop

While loop:

**Syntax:**

Initialization;

while (condition)

{

//Statements;

//iteration;

}

**For infinite loop:**

while (true)

{

printf (“deepak\n”);

}

For loop:

**Syntax:**

for(initialization ; condition ; iteration)

{

//statements;

}

**For infinite loop:**

for( ; 1 ; )

{

//statements;

}

**(or)**

for( ; ; )

{

//Statements;

}

Do while loop:

**Syntax:**

do

{

//statements;

//iteration;

}while(condition)

**Note:- continue** keyword work only for loop not switch but **break** keyword use bothcase loops &switch

# Programme:->CALCULATOR USING SWITCH

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Calculator\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*Note:-> to use this calculator use -lm when you compile this program

:-> command to use compile this programe is gcc filename.c -lm \*/

#include<stdio.h>

#include<stdlib.h>

#include<math.h>

int main()

{

int a,b,c,ch;

float d;

char cha;

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*WELCOME TO DOUNTLESS

CALCULATOR\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n(press enter):");

scanf("%c",&cha);

while(1)

{

printf("\n\t\tEnter the choice in the given below:\n");

printf("Addition:1\nSubstraction:2\nmultiplication:3\nDivision:4\nmodulation(Reminder):5\nSquare\_root:6\nexit:7\n\n:->");

scanf("%d",&ch);

switch(ch)

{

case 1:

printf("enter two no: ");

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

c=a+b;

printf("Addition of %d and %d is %d\n",a,b,c);

break;

case 2:

printf("enter two no: ");

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

c=a-b;

printf("Substraction of %d and %d is %d\n",a,b,c);

break;

case 3:

printf("enter two no: ");

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

c=a\*b;

printf("Multiplication of %d and %d is %d\n",a,b,c);

break;

case 4:

printf("enter two no: ");

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

d=(float)a/b;

printf("division of %d and %d is %6.2f\n",a,b,d);

break;

case 5:

printf("enter two no: ");

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

c=a%b;

printf("Addition of %d and %d is %d\n",a,b,c);

break;

case 6:

printf("enter a no.:");

break;

case 7:

exit(0);

default:

printf("enter a valid choice\n");

}

}

return 0;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Calculator\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**FUNCTION**

* **Function are used to divide a large programme into a smaller pieces.**
* **A function can be called multiple times to provide a reusability and modularity to the C programme**
* **Also called procedure and subroutine**
* Function is a piece of code to accomplish certain operation and it has a name for identification.
* A function is a group of statement that together a task. Every C programme has at least one function, which is **main(),** and all the most trivial programme can define additional function.
  + EX.:->>>>



# Function Declarations/prototype/signature:->

* + - * A function declaration just tells the compiler about the function name, function datatype, parameter and how to call the function.
      * A function tell the compiler about this existence.

# Function Calling:->

* + - * A function is called in order to be used.
      * While creating a function , you give define of what the function has to do. To use a function you will have to call that function to perform the defined task.

Ex:->

Functionname (pass by value /Reference)

# Function Definition:->

* + - * A function is define to get some task done.
      * Some function perform the desired operation without returning a value, in this case the return type is **void.**

## Explain:-> int fun(int,int);

* fun is a function with two argument as int and int and it will return int .

# Parameter & Argument:->

* + - * Function  **parameter** are the listed in the function definition.
      * Function Argument are the Real value passed to the function .

Parameter are initialize to the value of argument supplied.

# Call by value:->

* + - * This method copies the actual value of an argument into the formal parameter of the function. In this .

# Call by Reference:->

* + - * This method copies the address of an argument into the formal parameter of the function. In this case, the address is used to access the actual argument used in call. This means that change made to the parameter inside the function that affect on the argument.

## Recursion Function (Equivalent to loop)

* + - * Function calling itself is called recursion function
      * Best example to understanding of recursion function is factorial of any no.

Ex:->**Factorial of a number**

#include<stdio.h>

Int fact(int);

Int main()

{

Int num=4,Result;

Result=fact(num);

printf(“factorial of %d no. is:”,Result);

return 0;

}

Int fact(int a)

{

If (a==0);

return 1;

else

{

return (a\*fact(a-1));

}

## How to work recursion function :->



# **STORAGE CLASS**

* It tells about the allocation of variable in memory .

## MEMORY:->

* + - Basically memory are three type CPU register, RAM, secondary storage .

### RAM:-> **RAM(Random access memory) have Four Segment**

|  |
| --- |
| **STACK** |
| **HEAP** |
| **DATA** |
| **CODE/TEXT** |

## **Storage Class tells about four property:->**

* auto/Automatic
* register
* static
* global/extern

### Automatic storage class:-( auto)

* initial value/Default value->garbage value
* storage->stack
* scope->with in block

|  |
| --- |
| **STACK** |
| **HEAP** |
| **DATA** |
| **CODE/TEXT** |

* life->block end
* Note:-> optional (keyword:-> auto )

Ex:-> #include<stdio.h>

Int main()

{

Int x=5;

printf(“%d\n”,x);

{

Int x=2;

printf(“%d\n”,x);

//the x variable which is more local it will reference

}

printf (“%d\n”,x);

return 0;

}

Output:-> 5

2

5

### Register storage class: (register)

* initial value/Default value->garbage value
* storage->CPU Register/stack,( if register not available.)
* scope->with in block
* life->block end

|  |
| --- |
| **RAM** |

|  |
| --- |
| **CPU** |

|  |
| --- |
| **Register** |

|  |
| --- |
|  |

|  |
| --- |
| **acumulator** |

|  |
| --- |
| Logic calculator |

**programme will become fast slightly**

### Ex:->

#include<stdio.h>

Int main()

{ register int x=4;

Int y;

y=x++;

x--;

y=x+5;

printf(“%d %d”,x,y);

return 0;

}

Output:->4,9

### Static storage class:(static)

* initial value/Default value->Zero(0)
* storage->data segment
* scope->with in block
* life->programme end

|  |
| --- |
| **STACK** |
| **HEAP** |
| **DATA** |
| **CODE/TEXT** |

Ex:->

#include<stdio.h>

void F1(void);

int main()

{

F1();

F1();

return 0;

}

void F1()

{

static int i;

i++;

printf(“%d\n”,i);

}

Output:-> 1

2

### Global/Extern storage class: (global/extern)

* initial value/Default value-> Zero(0)
* storage-> data segment
* scope->across the block
* life-> programme end

|  |
| --- |
| **STACK** |
| **HEAP** |
| **DATA** |
| **CODE/TEXT** |

Ex:-1

#include<stdio.h>

void f1();

void f2();

Int x;//globally

Int main()

{

printf(“%d\n”,x);

f1();

f2();

print(“%d\n”,x);

return 0;

}

void f1()

{

x++;

printf(“%d\n”,x);

}

void f2()

{

static int x=5;

x++;

printf(“%d\n”,x);

}

Output:-> 0

1

6

1

Ex:2-

#include<stdio.h>

void g1();

Int main()

{

extern int x;

printf(“%d\n”,x);

g1();

printf(“%d\n”,x);

return 0;

}

Int x;

void g1()

{

x++;

printf(“%d\n”,x);

}

Output:-> 0

1

1

# **The extern Storage Class**

* The extern storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use **'extern'**, the variable cannot be initialized as all …..
* it does is point the variable name at a storage location that has been previously defined.
* When you have multiple files and you define a global variable or function, which will be used in other files also, then extern will be used in another file to give reference of defined variable or function.
* Just for understanding, extern is used to declare a global variable or function in another file.

**The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.**

**First File: main.c**

#include int count ;

extern void write\_extern();

main()

{

count = 5;

write\_extern();

}

**Second File: support.c**

#include extern int count;

void write\_extern(void)

{

printf("count is %d\n", count);

}

Here, extern keyword is being used to declare count in the second file where as it has its definition in the first file, main.c.

Now, compile these two files as follows: $gcc main.c support.c

This will produce a.out executable program, when this program is executed, it produces the following result: 5

# Pre-Processor:->

* pre-processor is a program which performs before compiler.
* Pre-Processor is a system software.
* Pre-Processor only notice **#** start symbol.
* **#** is called Pre-Processor directive.
* EachPre-Processing directive must be on its own line.
* The word after **#** is called Pre-Processor command.
* **Include**  is one of the most popular Pre-Processor command
* It can be used include any file content to your source file.

## #define

* The # define directive define an identifier and a character sequence (a set of character ) that will be substituted for the identifier each time it is encountered in the source file.
* Ex:-> #define PI 3.14 // here **PI** is a macro templates and **3.14**  is a macro expansion
* The first two statements in the programme are called macro definitions. So the ‘macro templates’ **PI** are replaced by ‘macro expansion’ **3.14** 
  + **Ex:->**

#include<stdio.h>

#define PI 3.14

Int main()

**{**

Int r;

float A;

printf(“enter radius of a circle”);

scanf(“%d”,&r);

A=PI\*r\*r;

printf(“Area of acircle is:%6.2f”,A);

return 0;

**}**

## Define Macro like a functions:->

#include<stdio.h>

#define sum(a,b) a+b

Int main()

{

printf(“sum is %d\n”,sum(4,2));

return 0;

}

Output-> **sum(a,b) -> a+b**

**sum(4,2)->4+2 :->>>6**

Ex:->2

#include<stdio.h>

#define sum(a,b) a\*b

Int main()

{

printf(“product is :%d”,sum(3+2 , 4-6));

return 0;

}

Output-> **sum(a,b) -> a\*b**

**sum(3+2 , 4-6 )->3+2\*4-6 :->>>5**

Ex:->3

#include<stdio.h>

#define sum(a,b) a\*b

Int main()

{

printf(“product is :%d”,sum((3+2) ,( 4-6)));

return 0;

}

Output-> **sum(a,b) -> a\*b**

**sum( (3+2) , (4-6) )-> (3+2)\*(4-6):->>> -10**

## #undef

* It is used to undefine macros

Ex:->

#define CLOSE 0

…………………………..

…………………………. Identity macro

…………………………

#undef CLOSE

…………………… unidentity macros

* Int the Cprogramming language, the # undef directive tells the preprocessor to remove all definition for the specified macros.

## #**if, #else, #elif, #endif**

* As a file is being compiled, you can use these commands to cause certain lines of code to be include or not include(for compilation)

### #if

* + # if checks whether the value is true so, include the code in compilations file if not, that code is removed from the copy of the file given to the compiler prion to compilation and until the closing **#endif**

### #elif

* + It stands for ‘**else if’** to check more conditions and it goes in the middle of a conditional group and subdivides it.

### #else

* + the **#else** directives provides an alternate action when used with **#if, #ndef** or **#ifndef** directives.
  + The c source code that follows the **#else** statements when the condition for the #else statement when the condition for the **#if, #ifdef** and **#ifndef** directive evolutes to false.

### #ifdef ‘macro’

* + If the macro has been define by a **#define** value statement, then the code immediately following the command will be compiled

### #ifndef

* + If the macro has not been defined by a **#define** statement, then the code immediately following the command will be compiled

### ##

* + The **##** operator is used with the **#define** macro.
  + Using **##** concatenates what’s before the **##** with what’s after it .

Ex:->

#define ACTION(a,b) a##b+a\*b

Int main()

{

printf(“%d”,ACTION(3,4));

return 0;

}

Output:-> ACTION(3,4):->> 3##4+3\*4 -> 34+3\*4 -> 34+12 -> 46

Note:->

* + - If any year is modulous by 4==0 (y%4==0) , so this year is leap year
    - #include<stdio.h> //contain exit(0) and it end the programme

# Basic of pointers:->

**Int X=5**

x

5

**2048**

**X** is a name of memory location

**5** is the content of memory block

**2048** is the Address of the memory block

Ex:-> main()

{

Int x=5;

printf(“%d\n”,x);

printf(“%d”,&x);

}

Output:-> 5

2048

## Address of operator :->

* & is known as address of operator
* It is an unary operator
* Oprands must be name of the variable
* & give address number of variable
* & is also known as referencing operator

# Indirection operator

* \* is a indirection operators
* It is also known as dereferencing operator
* It is an unary operator
* It takes address as an argument
* \* return the content whose address is argument

Ex:-> main()

{

Int x=5;

Printf(“%d\n”,x);

Printf(“%d\n”,&x);

Printf(“%d”, \*&x);

}

Note:-> reference %u to print address because address is not negative.

x

5

**2048**

Ex:->1

Int x=5

&x=7

Output:->L value error because &x means 2048 that means 2048=7 not allowed because its constant.

Ex:->2

Int \*j ,x=5;

J=&x;

* J is not an ordinary variable like any other intiger variable
* J is a variable which contains the address of another variable.

Ex:-> #include<stdio.h>

{

Int x=5,\*j;

J=&x;

Printf(“%d %u\n”,x,j);

Printf(“%d %u”,\*j,&x);

Printf(“%u”,\*&j);

}

Output:-> 5 2048

5 2048

2048

## Base Address

Int a , \*j;

J a

1000

1000 1001

float b , \*k;

k b

2000

2000 2001 2002 2003

char c , \*r;

r c

3000

3000

J=&a,k=&b,r=&c;

# Pointer

* Pointer is a variable that contain address of the another variable
* Pointer always consume 4 bytes in memory
* So it is a special variable and it is of 4 byte variable

## Extended Concept of Pointer :->

x p q r

5 1000 2000 3000

1000 2000 3000 4000

void main()

{

Int x=5 , \*p ,\*\*q ,\*\*\*r;

p=&x;

q=&p;

r=&q;

\*\*\*r=7;// that means x=7;

}

Ex:->

p q r

6 0x100 0x200

0x100 0x200 0x300

Int p=6;

Int \*q=&p;

Int \*\*r=&q;

printf(“%d\n”,p);

p=6 &p=0x100

q=0x100 &q=0x200

r=0x200 &r=0x300

\*p=illigle \*\*p=illigle

\*q=6 \*\*q=illigle

\*r=0x100 \*\*r=6

\*&p=6

\*&q=0x100

\*&r=0x200

## Pointers Arithmatic

* We can not add,multiply,divide two addresses(Substraction is possible)
* We can not multiply an intiger to an address and similarly we cannot divide an address with an intiger value
* We can only add or substraction intiger to /form as address

**Formulae:->Pointer + n =pointer + sizeof(type of pointer) \* n**

Int \*p;

p

1000

p+1 = 1004

p+2 = 1008

p+3 = 10012

**substraction=Pointer -n =>pointer - sizeoff(type of pointer ) \* n**

1000 – 4 \*2

1000 – 8 = 1992

p-1 = 996

p-2 = 992

### **Pointer1 – pointer2 = Literal substraction / size of (type of pointer)**

* 1020 – 1000 => 20(address)/4(size of memory block) = 5 (make five memory block)

# Dangling pointer

* A pointer without pointing of any location
* Ex:-> int \*p;

# NULL pointer

* Null pointer is a pointer pointing to a NULL

EX:-> int \*p = 0

Int \*p = NULL

## Note:->

* Where Null is a macro and define in **stdio.h** as **a # define (void \*) Null 0**
* its help in reducing the memory leakage

# Generic pointer

* void \* can point to any data type

int a=10;

char ch=’r’;

float b= 10.55;

void \* ptr;

ptr=&a;

ptr=&ch;

ptr=&b;

printf(“%d\n”,\*(int)ptr);

printf(“%c\n”,\*(char)ptr);

printf(“%f\n”,\*(float)ptr);

### Note:->

* External typecasting / Explicit typecasting is \*(float) type….
* If we are not properly typecasting a pointer compiler generate warning…

# **Arrays**

* Array is a linear collection of similar element.
* Array is also known as subscript variable.
* Array is a group of variable

EX:->

index no. Int a[10]; Array

a

**0 1 2 3 4 5 6 7 8 9**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **5** |  |  |  |  |  | **2** |  |  |  |

**a[0]=5 // index of array**

**a[6]=2**

ex:->

Calculate avg. of no. 1 to 10

#include<stdio.h>

Int main()

{

Int a[10],i,sum=0;

float avg;

printf(“enter 10 numbers:->”);

for(i=0;i<=9;i++)

scanf(“%d”,&a[i]);

for(i=0;i<9;i++)

sum=sum+a[i];

avg=sum/10.0;

printf(“average of 10 numbers is :->%6.2f”,avg);

return 0;

}

# Declaration of Arrays

* Int a[]; Error a

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| G.V | G.V | G.V | G.V | G.V |

* Int a[5];
* Int a[5]={9,2,5,6,7}; a

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 9 | 2 | 5 | 6 | 7 |

* Int a[]={9,2,5,6,7};
* Int a[5]={3,4,5,8,7,4,9,4};//error two many argument
* Int a[5]={4,5,7} a

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 4 | 5 | 7 | 0 | 0 |

* Int a[5];
  + a[0]=9 a

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 9 | G.V | G.V | G.V | G.V |

**Ex:-> Add two 1D array**

#include<stdio.h>

Int main()

{

Int a[3],b[3],sum=0,i;

printf(“enter number in first array:->”);

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

scanf(“%d”,&a[i]);

printf(“enter numbers for second array:->”);

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

scanf(“%d”,&b[i]);

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

sum=sum+a[i]+b[i];

printf(“sum of two array is :->%d”,sum);

return 0;

}

**2D Arrays(**Two dimensional Array**)**

* Array of Arrays

1D Array

Ex:->Int a[6];

a

0 1 2 3 4 5

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |

Int b[2][3];

b

0 1

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  |   0 1 2 | |  |  |  | | --- | --- | --- | |  |  |  |   0 1 2 |

(or)

b

b[0][1]

0 1 2 b[0][2]

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |  |
| --- | --- | --- |
|  |  |  |

0

0 1 2

1

b[1][2]

b[1][0]

**Ex:-> Add two Matrices**

#include<stdio.h>

int main()

{

int a[3][3],b[3][3],c[3][3],i,j;

printf("enter 9 element in first aray:->");

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

{

for(j=0;j<=2;j++)

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

}

printf("enter 9 element in second array:->");

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

{

for(j=0;j<=2;j++)

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

}

printf("Sum of Matrix is:->\n");

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

{

for(j=0;j<=2;j++)

{

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

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

}

printf("\n");

}

return 0;

}

# **Array Using Pointer**

* Array always consume memory location in contiguous fashion

Int a[8];

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

* Pointer when incremented, always point to immediately next block of its own type.

Ex:->

#include<stdio.h>

Int main()

{

Int I,a[5],\*p;

p=&a[0];

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

scanf(“%d”,p+i);

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

printf(“%d\n”,\*(p+i));

return 0;

}

* Name of the array is base address of it

Base address

* + - * + a

|  |  |  |  |
| --- | --- | --- | --- |
| 10 | 20 | 45 | 32 |

1000 1004 1008 1012

* While acting with pointer/address in a loop it iterates one location to whome it belong.

Note:->1

* If the array is created in a programm so on Runtime we cannot change size of array during execution of programm
* Otherwise it give error
* So for overcome this limitation C language provide **malloc()** and **calloc()**

Note:->2

* \*(b+i) same as \*(i+b) , which is b[i] same as i[b]
* Array once is declared is its base address cant change by operation ++ if it dose so , It would be unable to remember the beginning of the array.

# String:->

* Sequence of character terminated at null character
* ASCII value of null character is 0 (Zero)

## Input/Output Operation with string

S

0 1 2 3 4 5 6 7

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| D | E | E | P | A | K | \0 |  |

1000 1001 1002 1003 1004 1005 1006

Note:->

* ‘\0’ is essential to put at the end of string because compiler don’t know where the string is terminated.
* Pointer variable consume 4 byte memory always so if you increment 1 on address of pointer variable it will move …
* Ex:-> char q=’D’;

char \*p=&q // suppose address of q is 1000

p+1=1000+1=1001

char \*\*pt=&p

/\* suppose address of p (pointer variable) is 2000\*/

pt+1= 2000+1=2004

Ex:->1

Int main()

{ char s[]={ ‘D’ , ’E’ , ’E’ , ’P’ , ’A’ , ’K’ , ’\0’ }

Int i;

for(i=0;s[i]!=’\0’;i++)

printf(“%c”,s[i]);

return 0;

}

Ex:->2

Int main()

{ char s[]={ ‘D’ , ’E’ , ’E’ , ’P’ , ’A’ , ’K’ , ’\0’ }

printf( “%s”,s); // or printf(“%s”,&s[0]);

return 0;

}

E:->3

Int main()

{

Char s[]=”deepak”;

/\* compiler put ‘\0’ automatically at the end of string\*/

puts(s); // or puts(&s[0]);

\\* puts() function works as printf() and puts \n(next line) automatically\*\

return 0;

}

Ex:->4

Int main()

{

char s[10];

printf(“enter name:->”);

scanf(“%s”,s);

printf(“%s”,s);

return 0;

}

Note:->

* **scanf()** not allow multiple words for input because if you enter **Dennis Ritchie** it take Dennis as a string because scanf() function understand delimiter after first word (space ,enter ,tab) are delimiter
* ….. so, we use **gets()** function instead of **scanf()** function because **gets()** ignore delimiter

Ex:->5

Int main()

{

char s[10];

printf(“enter name:->”);

gets(s); // or gets(&s[0])

puts(s);

return 0;

}

### Handling a Multiple String

char s[3][10]={“deepak”,”aman”,”amit”};

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| d | e | e | p | a | k | \0 |  |  |  |

0

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| a | m | a | n | \0 |  |  |  |  |  |

1

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| a | m | i | t | \0 |  |  |  |  |  |

2

#include<stdio.h>

Int main()

{

char s[3][10];

int i;

printf(“enter three string :->”);

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

gets(s[i][0]);

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

puts(s[i][0]);

return 0;

}

# Important String Related Function:->

* Strlen():-> it give string length
* Strrev():-> it reverse a string
* Strlwr():->it convert lowercase character
* Strupr():->it convert uppercase character
* Strcpy():-> it copy string in array
* Strcmp():->it compare two string

Ex:-> strcmp(“dee”,”pak”) so it return int value of ascii difference of corresponding mismatch character

* Strcat():->string concatenation
  + Ex;->
    - S[10]=”hello”;
    - Strcat(s,”student”);

# **SMA**:-Static Memory Allocation

* Static variable………In General , static memory allocation is to decide how much memory allocate in memory at compile time, before the associated programme is executed , Unlike dynamic memory allocation
* Ex:-> int a; //SMA

# **DMA:-**Dynamic MemoryAllocation

* Dynamic Memory Allocation (or) Automatic Memory Alloction where memory is allocated as required at run time.
* Ex:->
  + malloc()
  + calloc()
  + realloc()
  + free()

## malloc()

* malloc() are the library function that the allocate memory dynamically. It means that memory is allocated during runtime(execution of programm) from the heap segment.
  + Ex:->int \*p

p=(typecast)**malloc**(size of block)

* It return a point of void(void \*) which can be casted int pointer of any form (int \*) etc.
* Ex:->p=(int\*)**malloc**(100\*sizeof(int));
  + Since the size of int is 4 byte and this statement will allocated 400 bytes of memory
  + It initialize each block with default garbage value

## calloc()

* calloc () function stands for ‘contiguous allocation’
* calloc() are the library function that the allocate memory dynamically.
* This function used to allocate multiple block of memory .
* It is a dynamic memory allocation function which is used to allocate the memory to complex data\_structure and such as array and structure
* The functions takes in two parameters that collectively specify the amount of memory to be allocated
  + Ex:-> int \*ptr
    - ptr=(typecast)calloc(5,2);
    - 5 i**s no. of memory block**
    - 2 **is size of memory block**
* it initialize each block with default value 0(zero)

# malloc() VS calloc()

* malloc() takes single argument(memory required in bytes)
* calloc() needs two arguments(no of memory block , size of memory block)
  + Ex:-> int \*ptr
    - ptr=(typecast)calloc(5,2);
    - 5 i**s no. of memory block**
    - 2 **is size of memory block**
* malloc() does not initialize the memory allocated
* calloc() initialize the allocated memory to zero

# realloc() and free()

* The function **realloc()** is used to resize the memory block which is allocated by malloc() and calloc()

Here is the syntax of realloc() in c language

Ex:-> (void\*)realloc(void\*pointer , size)

Typecast address of block resize no.

* The function free() is used to de-allocate the memory allocated by function malloc(),calloc(),realloc() etc.

* The argument of the function free() is the pointer of the memory which is to be freed

# Structure

* Structure is a way to group of variable
* Structure is a collection of dissimilar element as well as similar element
* Defining structure means creating new datatype also called as user define datatype

## Define Structure datatype

struct book

{

//variable declaration;

};

* No memory consume for definition of structure because it is a datatype and but variable consume memory according to datatype

Ex:->

void main()

{ **variable consume memory according to datatype**

float x;

}

**Its define**

## **As well as**

struct date

{ **its define but memory not consume because it is a datatype same as float**

Int x,y,z

};

* To make user datatype **struct** keyword use before make datatype

## Declare datatype

* It will be must to declare datatype to use **struct** keyword before every datatype

struct date a; keyword datatype variable

* We also declare Variable on define datatype

struct book

{

//d,y,z;

}x;

Here x is a variable

Note:->

* Structure element(member variable) are always arranged in contiguous memory locations.

# Assigning Variable in structure

Struct date

{

Int x,y,z;

};

void main()

{

struct date D1,d2={25,7,2015};

(or) d2

D1.x=2;

x

y

z

D1.y=4;

27

7

2015

D1.z=2017;

* A structure elements are accessed using **‘.’** Operator

// we want to copy d2 variable data to d1 variable

d1=d2

(or)

printf(“enter today date by formate d/m/yr”);

scanf(“%d%d%d”,&today.x,&today.y,&today.z);

printf(“today date is:->%d/%d/%d”,today.x,today.y,today.z);

ex;->

#include<stdio.h>

struct book

{

Int bookid;

char title[20];

float price;

};

struct book input();

void display(struct book);

int main()

{

struct book b1;

b1=input();

display(b1);

return 0;

}

struct book input();

{

Struct book b;

printf(“enter book id ,title and price:->”);

scanf(“%d%s%f”,&b.bookid,b.title,&b.price);

return b;

}

void display(struct book c)

{

printf(“book id=%d\ntitle=%s\nprice=%f”,c.bookid,c.title,c.price);

}

# Note:->one structure can be nested within another structure

struct part

{

char type;

int qty;

};

Struct vehicle

{

Char maruti[20];

Struct part bolt;

};

Struct vehicle b1;

b1.bolt.qty=200;

# Note :->

* To access structure element through a structure variable we use the ‘.’ Operator .Whereas to access structure element through a pointer to a structure we use the

‘->’ operator

Ex:->

struct book

{

char name[25];

int no;

};

struct book b1={“c exploring”,2020012006};

struct book \*b2;

printf(“%s %d”,b.name,b.no);

b2=&b1;

printf(“%s %d”,b2->name,b2->no);

## Structure padding:->

* In order to align the data in memory,  one or more empty bytes (addresses) are inserted (or left empty) between memory addresses which are allocated for other structure members while memory allocation. This concept is called structure padding.
* Architecture of a computer processor is such a way that it can read 1 word (4 byte in 32 bit processor) from memory at a time.
* To make use of this advantage of processor, data are always aligned as 4 bytes package which leads to insert empty addresses between other member’s address.
* Because of this structure padding concept in C, size of the structure is always not same as what we think.

       For example, please consider below structure that has 5 members.

struct student

{

       int id1;

       int id2;

       char a;

       char b;

       float percentage;

};..

* As per C concepts, int and float datatypes occupy 4 bytes each and char datatype occupies 1 byte for 32 bit processor. So, only 14 bytes (4+4+1+1+4) should be allocated for above structure.
* But, this is wrong.  Do you know why?
* Architecture of a computer processor is such a way that it can read 1 word from memory at a time.
* 1 word is equal to 4 bytes for 32 bit processor and 8 bytes for 64 bit processor. So, 32 bit processor always reads 4 bytes at a time and 64 bit processor always reads 8 bytes at a time.
* This concept is very useful to increase the processor speed.
* To make use of this advantage, memory is arranged as a group of 4 bytes in 32 bit processor and 8 bytes in 64 bit processor.
* Below C program is compiled and executed in 32 bit compiler. Please check memory allocated for structure1 and structure2 in below program.

#### **EXAMPLE PROGRAM FOR STRUCTURE PADDING IN C LANGUAGE:**

#include <stdio.h>

#include <string.h>

/\*  Below structure1 and structure2 are same.

    They differ only in member's allignment \*/

struct structure1

{

       int id1;

       int id2;

       char name;

       char c;

       float percentage;

};

struct structure2

{

       int id1;

       char name;

       int id2;

       char c;

       float percentage;

};

int main()

{

    struct structure1 a;

    struct structure2 b;

    printf("size of structure1 in bytes : %d\n",

            sizeof(a));

    printf ( "\n   Address of id1        = %u", &a.id1 );

    printf ( "\n   Address of id2        = %u", &a.id2 );

    printf ( "\n   Address of name       = %u", &a.name );

    printf ( "\n   Address of c          = %u", &a.c );

    printf ( "\n   Address of percentage = %u",

                   &a.percentage );

    printf("   \n\nsize of structure2 in bytes : %d\n",

                   sizeof(b));

    printf ( "\n   Address of id1        = %u", &b.id1 );

    printf ( "\n   Address of name       = %u", &b.name );

    printf ( "\n   Address of id2        = %u", &b.id2 );

    printf ( "\n   Address of c          = %u", &b.c );

    printf ( "\n   Address of percentage = %u",

                   &b.percentage );

    getchar();

    return 0;

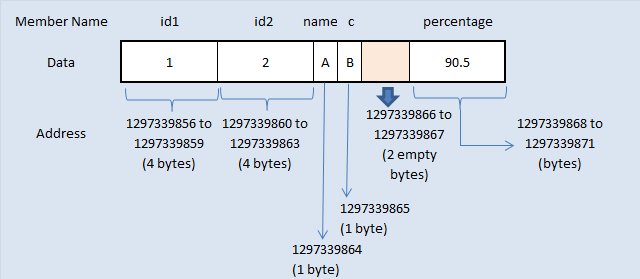
}

OUTPUT:->

**size of structure1 in bytes : 16**  
Address of id1 = 1297339856  
Address of id2 = 1297339860  
Address of name = 1297339864  
Address of c = 1297339865  
Address of percentage = 1297339868  
**size of structure2 in bytes : 20**  
Address of id1 = 1297339824  
Address of name = 1297339828  
Address of id2 = 1297339832  
Address of c = 1297339836  
Address of percentage = 1297339840

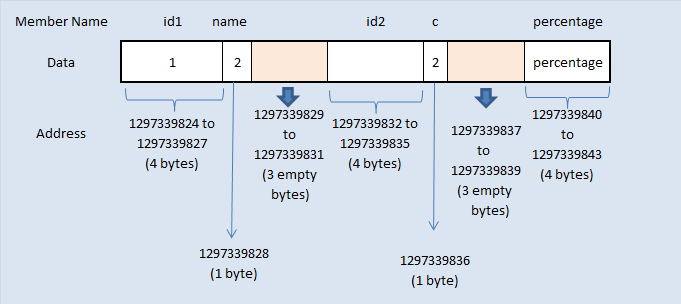
#### **STRUCTURE PADDING ANALYSIS FOR ABOVE C PROGRAM:**

#### **MEMORY ALLOCATION FOR STRUCTURE1:**



* In above program, memory for structure1 is allocated sequentially for first 4 members.
* Whereas, memory for 5th member “percentage” is not allocated immediate next to the end of member “c”.
* There are only 2 bytes remaining in the package of 4 bytes after memory allocated to member “c”.
* Range of this 4 byte package is from 1297339864 to 1297339867.
* Addresses 1297339864  and 1297339865 are used for members “name and c”. Addresses 1297339866  and 1297339867 only is available in this package.
* But, member “percentage” is datatype of float and requires 4 bytes. It can’t be stored in the same memory package as it requires 4 bytes. Only 2 bytes are free in that package.
* So, next 4 byte of memory package is chosen to store percentage data which is from 1297339868 to 1297339871.
* Because of this, memory 1297339866 and 1297339867 are not used by the program and those 2 bytes are left empty.
* So, size of structure1 is 16 bytes which is 2 bytes extra than what we think. Because, 2 bytes are left empty.

#### **MEMORY ALLOCATION FOR STRUCTURE2:**



* Memory for structure2 is also allocated as same as above concept. Please note that structure1 and structure2 are same. But, they differ only in the order of the members declared inside the structure.
* 4 bytes of memory is allocated for 1st structure member “id1” which occupies whole 4 byte of memory package.
* Then, 2nd structure member “name” occupies only 1 byte of memory in next 4 byte package and remaining 3 bytes are left empty. Because, 3rd structure member “id2” of datatype integer requires whole 4 byte of memory in the package. But, this is not possible as only 3 bytes available in the package.
* So, next whole 4 byte package is used for structure member “id2”.
* Again, 4th structure member “c” occupies only 1 byte of memory in next 4 byte package and remaining 3 bytes are left empty.
* Because, 5th structure member “percentage” of datatype float requires whole 4 byte of memory in the package.
* But, this is also not possible as only 3 bytes available in the package. So, next whole 4 byte package is used for structure member “percentage”.
* So, size of structure2 is 20 bytes which is 6 bytes extra than what we think. Because, 6 bytes are left empty.

#### **HOW TO AVOID STRUCTURE PADDING IN C LANGUAGE?**

* #pragma pack ( 1 ) directive can be used for arranging memory for structure members very next to the end of other structure members.
* VC++ supports this feature. But, some compilers such as Turbo C/C++ does not support this feature.
* Please check the below program where there will be no addresses (bytes) left empty because of structure padding.

#### **EXAMPLE PROGRAM TO AVOID STRUCTURE PADDING IN C:**

#include <stdio.h>

#include <string.h>

/\*  Below structure1 and structure2 are same.

    They differ only in member's allignment \*/

#pragma pack(1)

struct structure1

{

       int id1;

       int id2;

       char name;

       char c;

       float percentage;

};

struct structure2

{

       int id1;

       char name;

       int id2;

       char c;

       float percentage;

};

int main()

{

    struct structure1 a;

    struct structure2 b;

    printf("size of structure1 in bytes : %d\n",

                   sizeof(a));

    printf ( "\n   Address of id1        = %u", &a.id1 );

    printf ( "\n   Address of id2        = %u", &a.id2 );

    printf ( "\n   Address of name       = %u", &a.name );

    printf ( "\n   Address of c          = %u", &a.c );

    printf ( "\n   Address of percentage = %u",

                   &a.percentage );

    printf("   \n\nsize of structure2 in bytes : %d\n",

                   sizeof(b));

    printf ( "\n   Address of id1        = %u", &b.id1 );

    printf ( "\n   Address of name       = %u", &b.name );

    printf ( "\n   Address of id2        = %u", &b.id2 );

    printf ( "\n   Address of c          = %u", &b.c );

    printf ( "\n   Address of percentage = %u",

                   &b.percentage );

    getchar();

    return 0;

}

#### **OUTPUT:**

|  |
| --- |
| **size of structure1 in bytes : 14** Address of id1 = 3438103088 Address of id2 = 3438103092 Address of name = 3438103096 Address of c = 3438103097 Address of percentage = 3438103098 **size of structure2 in bytes : 14** Address of id1 = 3438103072 Address of name = 3438103076 Address of id2 = 3438103077 Address of c = 3438103081 Address of percentage = 3438103082 |

# UNIONS

* Union is a user-define datatype(very similar to structure)
* The difference between structure and unions lies in the fact that in structure, Each members. Has its own storage location, whereas members of union uses a single shared memory location.
* The single shared memory location is equal to size of its largest data members
* Defining Union means creating new datatype

Union is similar to structure, Except it allows you to define variables that shared storage space

* Below table will help you how to form a C union, declare a union, initializing and accessing the members of the union.

|  |  |
| --- | --- |
| **Using normal variable** | **Using pointer variable** |
| **Syntax:** union tag\_name { data type var\_name1; data type var\_name2; data type var\_name3; }; | **Syntax:** union tag\_name { data type var\_name1; data type var\_name2; data type var\_name3; }; |
| **Example:** union student { int  mark; char name[10]; float average; }; | **Example:** union student { int  mark; char name[10]; float average; }; |
| **Declaring union using normal variable:** union student report; | **Declaring union using pointer variable:** union student \*report, rep; |
| **Initializing union using normal variable:** union student report = {100, “Mani”, 99.5}; | **Initializing union using pointer variable:** union student rep = {100, “Mani”, 99.5}; report = &rep; |
| **Accessing union members using normal variable:** report.mark; report.name; report.average; | **Accessing union members using pointer variable:** report  -> mark; report -> name; report -> average; |

#### **EXAMPLE PROGRAM FOR C UNION:**

#include <stdio.h>

#include <string.h>

union student

{

            char name[20];

            char subject[20];

            float percentage;

};

int main()

{

    union student record1;

    union student record2;

    // assigning values to record1 union variable

       strcpy(record1.name, "Raju");

       strcpy(record1.subject, "Maths");

       record1.percentage = 86.50;

       printf("Union record1 values example\n");

       printf(" Name       : %s \n", record1.name);

       printf(" Subject    : %s \n", record1.subject);

       printf(" Percentage : %f \n\n", record1.percentage);

    // assigning values to record2 union variable

       printf("Union record2 values example\n");

       strcpy(record2.name, "Mani");

       printf(" Name       : %s \n", record2.name);

       strcpy(record2.subject, "Physics");

       printf(" Subject    : %s \n", record2.subject);

       record2.percentage = 99.50;

       printf(" Percentage : %f \n", record2.percentage);

       return 0;

}

OUTPUT:

Union record1 values example  
Name :  
Subject :  
Percentage : 86.500000;  
Union record2 values example  
Name : Mani  
Subject : Physics  
Percentage : 99.500000

#### **EXPLANATION FOR ABOVE C UNION PROGRAM:**

There are 2 union variables declared in this program to understand the difference in accessing values of union members.

**Record1 union variable:**

* “Raju” is assigned to union member “record1.name” . The memory location name is “record1.name” and the value stored in this location is “Raju”.
* Then, “Maths” is assigned to union member “record1.subject”. Now, memory location name is changed to “record1.subject” with the value “Maths” (Union can hold only one member at a time).
* Then, “86.50” is assigned to union member “record1.percentage”. Now, memory location name is changed to “record1.percentage” with value “86.50”.
* Like this, name and value of union member is replaced every time on the common storage space.
* So, we can always access only one union member for which value is assigned at last. We can’t access other member values.
* So, only “record1.percentage” value is displayed in output. “record1.name” and “record1.percentage” are empty.

**Record2 union variable:**

* If we want to access all member values using union, we have to access the member before assigning values to other members as shown in record2 union variable in this program.
* Each union members are accessed in record2 example immediately after assigning values to them.
* If we don’t access them before assigning values to other member, member name and value will be over written by other member as all members are using same memory.
* We can’t access all members in union at same time but structure can do that.

#### **NOTE:**

* We can access only one member of union at a time. We can’t access all member values at the same time in union.
* But, structure can access all member values at the same time. This is because, Union allocates one common storage space for all its members. Wher**e**as Structure allocates storage space for all its members separately.

#### **DIFFERENCE BETWEEN STRUCTURE AND UNION IN C:**

|  |  |
| --- | --- |
| **C Structure** | **C Union** |
| Structure allocates storage space for all its members separately. | Union allocates one common storage space for all its members. Union finds that which of its member needs high storage space over other members and allocates that much space |
| Structure occupies higher memory space. | Union occupies lower memory space over structure. |
| We can access all members of structure at a time. | We can access only one member of union at a time. |
| Structure example: struct student { int mark; char name[6]; double average; }; | Union example: union student { int mark; char name[6]; double average; }; |
| For above structure, memory allocation will be like below. int mark – 2B char name[6] – 6B double average – 8B  Total memory allocation = 2+6+8 = 16 Bytes | For above union, only 8 bytes of memory will be allocated since double data type will occupy maximum space of memory over other data types.  Total memory allocation = 8 Bytes |

# TYPEDEF:

Typedef is a keyword that is used to give a new symbolic name for the existing name in a C program. This is same like defining alias for the commands.

Consider the below structure.

struct student  
{  
         int mark [2];  
         char name [10];  
         float average;  
}

Variable for the above structure can be declared in two ways.

**1st way  :**

struct student record;       /\* for normal variable \*/  
struct student \*record;     /\* for pointer variable \*/

**2nd way :**

typedef struct student status;

* When we use “typedef” keyword before struct <tag\_name> like above, after that we can simply use type definition “status” in the C program to declare structure variable.
* Now, structure variable declaration will be, “status record”.
* This is equal to “struct student record”. Type definition for “struct student” is status. i.e. status = “struct student”

#### **AN ALTERNATIVE WAY FOR STRUCTURE DECLARATION USING TYPEDEF IN C:**

typedef struct student  
{  
         int mark [2];  
         char name [10];  
         float average;  
} status;

**To declare structure variable, we can use the below statements.**

status record1;                 /\* record 1 is structure variable \*/  
status record2;                 /\* record 2 is structure variable \*/

#### **EXAMPLE PROGRAM FOR C TYPEDEF:**

// Structure using typedef:

#include <stdio.h>

#include <string.h>

typedef struct student

{

  int id;

  char name[20];

  float percentage;

} status;

int main()

{

  status record;

  record.id=1;

  strcpy(record.name, "Raju");

  record.percentage = 86.5;

  printf(" Id is: %d \n", record.id);

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

  printf(" Percentage is: %f \n", record.percentage);

  return 0;

}

OUTPUT:

Id is: 1  
Name is: Raju  
Percentage is: 86.500000

* Typedef can be used to simplify the real commands as per our need.
* For example, consider below statement.
* typedef long long int LLI;
* In above statement, LLI is the type definition for the real C command “long long int”. We can use type definition LLI instead of using full command “long long int” in a C program once it is defined.

Ex:2

#include <stdio.h>

#include <limits.h>

int main()

{

   typedef long long int LLI;

   printf("Storage size for long long int data " \

          "type  : %ld \n", sizeof(LLI));

   return 0;

OUTPUT:

Storage size for long long int data type : 8

# ENUMERATOR:

* It give an opportunity to invent own datatype and define what value the variable of this datatype can take.

Ex:->1.0

enum month

{

Jan, feb, mar, apr, may, jun, jul, Aug, sep, oct, nov, dec.

};

Int main()

{

enum month m1,m2,m3;

m1=jul;

return 0;

}

* Internally compiler treats the enumerators as a intiger.
* Each value in the list of permissible values corresponds to an integer, starting with 0(zero). In this example 1.0, jan is stored as 0, feb is stored as 1 ,……………….dec stored as 11.
* We can initialize enumerators with different intiger values.

Ex:->1.1

enum bollean

{

false,true

};

* You can write any programm in c language without the help of enumerators but , enumerators help in writing clear codes and simplify programming.

Ex:->1.1

#include<stdio.h>

enum Boolean

{

false true

};

enum Boolean isEven(int);

Int main()

{

enum Boolean d;

int n;

printf(“entera number\n”);

scanf(“%d”,&n);

d=isEven(n);

if(d==true)

printf(“no. is even\n”);

else

printf(“no. is odd\n”);

return 0;

}

enum Boolean isEven(int x)

{

If(x%2==0)

return(true);

else

return(false);

}

# FILE HANDLING:

* File handling concept in c language is used for store a data permanently in computer. Using this concept we can store our data in secondary memory(**hard disk).**
* All file related function are available in stdio.h header file.

**Note:->**

* Data structure of file in define as **FILE** in standard I/O function. So all files should be declared as type **FILE**

a(variable in file)

HDD HDDHHXHghg copy of file

sscssc

3

0101010101010101010101

**FILE**

ptr (FILE pointer)

**FILE**

* FILE is a structure (non-primitive datatypr)

typedef struct

{

short level;

unsigned flags;

char fd;

unsigned char hold;

short bsize;

unsigned char \*buffer

unsigned char \*curp;

unsigned istemp;

short token;

}FILE;

FILE \*fp;

RAM

open() fp here fp declare a variable as a pointer to the datatype **FILE**

FILE

# **WHAT IS FILE?**

File is a collection of bytes that is stored on secondary storage devices like disk. There are two kinds of files in a system. They are,

1. Text files (ASCII)
2. Binary files

* Text files contain ASCII codes of digits, alphabetic and symbols.
* Binary file contains collection of bytes (0’s and 1’s). Binary files are compiled version of text files.

# BASIC FILE OPERATIONS IN C PROGRAMMING:

There are 4 basic operations that can be performed on any files in C programming language. They are,

1. Opening/Creating a file
2. Closing a file
3. Reading a file
4. Writing in a file
5. Let us see the syntax for each of the above operations in a table:

|  |  |
| --- | --- |
| **File operation** | **Declaration & Description** |
| **fopen()** – To open a file | fopen() function is used to open a file to perform operations such as reading, writing etc. In a C program, we declare a file pointer and use fopen() as below. fopen() function creates a new file if the mentioned file name does not exist.  FILE \*fp; fp=**fopen**(“filename”, ”‘mode”);  Where, fp – file pointer to the data type “FILE”. filename – the actual file name with full path of the file. mode – refers to the operation that will be performed on the file. Example: r, w, a, r+, w+ and a+. Please refer below the description for these mode of operations. |
| **fclose()**– To close a file | fclose() function closes the file that is being pointed by file pointer fp. In a C program, we close a file as below. **fclose**(fp); |
| **fgets()** – To read a file | fgets function is used to read a file line by line. In a C program, we use fgets function as below. **fgets** (buffer, size, fp);  where, buffer – buffer to  put the data in. size – size of the buffer fp – file pointer  fgets return a NULL value when it reads EOF |
| **fprintf()**– To write into a file | fprintf() function writes string into a file pointed by fp. In a C program, we write string into a file as below.fprintf (fp, “some data”); or fprintf (fp, “text %d”, variable\_name); |

# **MODE OF OPERATIONS PERFORMED ON A FILE IN C LANGUAGE:**

There are many modes in opening a file. Based on the mode of file, it can be opened for reading or writing or appending the texts. They are listed below.

|  |  |  |
| --- | --- | --- |
| **Mode** | **Meaning** | **Description** |
| **r** | Read | Only reading possible .not create file if not exists. |
| **w** | write` | Only wirtting possible. Create file if not exists otherwise erase the old content of file and open as a blank file. |
| **a** | Append | Only writing possible .create file if not exists, otherwise open file and write from the end of file(**do not erase the old content)** |
| **r+** | Reading +writing | R&W. create file if not exists, overwriting existing data.used for modifying content |
| **w+** | Reading + writing | R&W. create file if not exists. Erase old content |
| **a+** | Reading+ appending | R&W possible. create file if not exists.append content at the end of file. |

**Note:->Reading from a file means**

* Extracting data from a file to our programme variable
* This will no remove data from a file.

# **1. EXAMPLE PROGRAM FOR FILE OPEN, FILE WRITE AND FILE CLOSE IN C LANGUAGE:**

|  |
| --- |
| / \* Open, write and close a file : \*/  # include <stdio.h>  # include <string.h>    int main( )  {      FILE \*fp ;      char data[50];      // opening an existing file      printf( "Opening the file test.c in write mode" ) ;      fp = fopen("test.c", "w") ;      if ( fp == NULL )      {          printf( "Could not open file test.c" ) ;          return 1;      }      printf( "\n Enter some text from keyboard” \               “ to write in the file test.c" ) ;      // getting input from user      while ( strlen ( gets( data ) ) > 0 )      {          // writing in the file          fputs(data, fp) ;          fputs("\n", fp) ;      }      // closing the file      printf("Closing the file test.c") ;      fclose(fp) ;      return 0;  } |

#### **OUTPUT:**

|  |
| --- |
| Opening the file test.c in write mode  Enter some text from keyboard to write in the file test.c Hai, How are you? Closing the file test.c |

# **2. EXAMPLE PROGRAM FOR FILE OPEN, FILE READ AND FILE CLOSE IN C LANGUAGE:**

This file handling C program illustrates how to read the contents of a file. Assume that, a file called “test.c” contains the following data “Hai, How are you?”. Let’s read this data using following C program.

/\* Open, Read and close a file: reading string by string \*/

# include <stdio.h>

int main( )

{

         FILE \*fp ;

         char data[50] ;

         printf( "Opening the file test.c in read mode" ) ;

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

         if ( fp == NULL )

         {

                 printf( "Could not open file test.c" ) ;

                 return 1;

         }

         printf( "Reading the file test.c" ) ;

         while( fgets ( data, 50, fp ) != NULL )

         printf( "%s" , data ) ;

         printf("Closing the file test.c") ;

         fclose(fp) ;

         return 0;

}

#### **OUTPUT:**

|  |
| --- |
| Opening the file test.c in read mode  Reading the file test.c  Hai, How are you?  Closing the file test.c |

# **INBUILT FUNCTIONS FOR FILE HANDLING IN C LANGUAGE:**

C programming language offers many inbuilt functions for handling files. They are given below. Please click on each function name below to know more details, example programs, output for the respective file handling function.

|  |  |
| --- | --- |
| **File handling functions** | **Description** |
| [fopen ()](http://fresh2refresh.com/c-programming/c-file-handling/fopen-fclose-gets-fputs-functions-c/) | fopen () function creates a new file or opens an existing file. |
| [fclose ()](http://fresh2refresh.com/c-programming/c-file-handling/fopen-fclose-gets-fputs-functions-c/) | fclose () function closes an opened file. |
| [getw ()](http://fresh2refresh.com/c-programming/c-file-handling/getw-putw-functions-c/) | getw () function reads an integer from file. |
| [putw ()](http://fresh2refresh.com/c-programming/c-file-handling/getw-putw-functions-c/) | putw () functions writes an integer to file. |
| [fgetc ()](http://fresh2refresh.com/c-programming/c-file-handling/fgetc-function-c/) | fgetc () function reads a character from file. |
| [fputc ()](http://fresh2refresh.com/c-programming/c-file-handling/fputc-function-c/) | fputc () functions write a character to file. |
| [gets ()](http://fresh2refresh.com/c-programming/c-file-handling/fopen-fclose-gets-fputs-functions-c/) | gets () function reads line from keyboard. |
| [puts ()](http://fresh2refresh.com/c-programming/c-file-handling/puts-function-c/) | puts () function writes line to o/p screen. |
| [fgets ()](http://fresh2refresh.com/c-programming/c-file-handling/fgets-function-c/) | fgets () function reads string from a file, one line at a time. |
| [fputs ()](http://fresh2refresh.com/c-programming/c-file-handling/fopen-fclose-gets-fputs-functions-c/) | fputs () function writes string to a file. |
| [feof ()](http://fresh2refresh.com/c-programming/c-file-handling/feof-function-c/) | feof () function finds end of file. |
| [fgetchar ()](http://fresh2refresh.com/c-programming/c-file-handling/fgetchar-function-c/) | fgetchar () function reads a character from keyboard. |
| [fprintf ()](http://fresh2refresh.com/c-programming/c-file-handling/fscanf-fprintf-ftell-rewind-functions-c/) | fprintf () function writes formatted data to a file. |
| [fscanf ()](http://fresh2refresh.com/c-programming/c-file-handling/fscanf-fprintf-ftell-rewind-functions-c/) | fscanf () function reads formatted data from a file. |
| [fputchar ()](http://fresh2refresh.com/c-programming/c-file-handling/fputchar-function-c/) | fputchar () function writes a character onto the output screen from keyboard input. |
| [fseek ()](http://fresh2refresh.com/c-programming/c-file-handling/fseek-seek_set-seek_cur-seek_end-functions-c/) | fseek () function moves file pointer position to given location. |
| [SEEK\_SET](http://fresh2refresh.com/c-programming/c-file-handling/fseek-seek_set-seek_cur-seek_end-functions-c/) | SEEK\_SET moves file pointer position to the beginning of the file. |
| [SEEK\_CUR](http://fresh2refresh.com/c-programming/c-file-handling/fseek-seek_set-seek_cur-seek_end-functions-c/) | SEEK\_CUR moves file pointer position to given location. |
| [SEEK\_END](http://fresh2refresh.com/c-programming/c-file-handling/fseek-seek_set-seek_cur-seek_end-functions-c/) | SEEK\_END moves file pointer position to the end of file. |
| [ftell ()](http://fresh2refresh.com/c-programming/c-file-handling/fscanf-fprintf-ftell-rewind-functions-c/) | ftell () function gives current position of file pointer. |
| [rewind ()](http://fresh2refresh.com/c-programming/c-file-handling/fscanf-fprintf-ftell-rewind-functions-c/) | rewind () function moves file pointer position to the beginning of the file. |
| [getc ()](http://fresh2refresh.com/c-programming/c-file-handling/getc-putc-functions-c/) | getc () function reads character from file. |
| [getch ()](http://fresh2refresh.com/c-programming/c-file-handling/getch-function-c/) | getch () function reads character from keyboard. |
| [getche ()](http://fresh2refresh.com/c-programming/c-file-handling/getche-function-c/) | getche () function reads character from keyboard and echoes to o/p screen. |
| [getchar ()](http://fresh2refresh.com/c-programming/c-file-handling/putchar-getchar-function-c/) | getchar () function reads character from keyboard. |
| [putc ()](http://fresh2refresh.com/c-programming/c-file-handling/getc-putc-functions-c/) | putc () function writes a character to file. |
| [putchar ()](http://fresh2refresh.com/c-programming/c-file-handling/putchar-getchar-function-c/) | putchar () function writes a character to screen. |
| [printf ()](http://fresh2refresh.com/c-programming/c-file-handling/printf-scanf-functions-c/) | printf () function writes formatted data to screen. |
| [sprinf ()](http://fresh2refresh.com/c-programming/c-file-handling/sprintf-function-c/) | sprinf () function writes formatted output to string. |
| [scanf ()](http://fresh2refresh.com/c-programming/c-file-handling/printf-scanf-functions-c/) | scanf () function reads formatted data from keyboard. |
| [sscanf ()](http://fresh2refresh.com/c-programming/c-file-handling/sscanf-function-c/) | sscanf () function Reads formatted input from a string. |
| [remove ()](http://fresh2refresh.com/c-programming/c-file-handling/remove-function-c/) | remove () function deletes a file. |
| [fflush ()](http://fresh2refresh.com/c-programming/c-file-handling/fflush-function-c/) | fflush () function flushes a file. |

**NOTE:->**

* **fclose()** can close only one file at a time. If you want to close three file ,we must call fclose three time.if we wish to close all the files through one call, we can make use of a function called **fcloseall()** . this closes all the files that are currently open, except the standard files like stdin,stdout,etc.

# fprintf():-

* fprintf() function is used to write formatted output to the specified stream.
* Int fprintf(FILE\*stream,const char \*formate[,argument,…..]);
* fprintf(fp,”sum of %d and %d is %d”,a,b,c);

**write a programm to write content to a file . use fprintf() to write content to the file.**

#include<stdio.h>

#include<stdlib.h>

Int main()

{

FILE \*fp;

Int a,b;

fp=fopen(“test.c”,”w”);

if(fp==NULL)

{

printf(“cannot open”);

exit(1);

}

printf(“enter two number:->”);

scanf(“%d%d”,&a,&b);

fwrite(fp,”%d%d”,a,b);

fclose(fp);

return 0;

}

# fscanf():->

* **fscanf()** function is used to read formatted content from a file.
* int fscanf(FILE\*stream,const char \*format,….);
* reads data from the stream and stores them accordingly to the parameter formate into the location pointed by the additional arguments
* **write a program to read content from a file and display on the screen . use fscanf() to the read content from the file**

#include<stdio.h>

#include<stdlib.h>

Int main()

{

Int a,b;

FILE \*fp;

fp=fopen(“test.c”,”r”);

if(fp==NULL)

{

printf(“cannot open”);

exit(1);

}

fscanf(fp,”%d%d”,&a,&b);

printf(“%d%d”,a,b);

fclose(fp);

return 0;

}

# fwrite():->

* fwrite() function is used to write content to the file in binary mode
* int fwrite(void \*buffer, int size , int count , FILE \*ptr);

**write a programme to write content to a file . use fwrite to write content to the file.**

#include<stdio.h>

#include<stdlib.h>

struct book

{

Int bookid;

char title[20];

float price;

};

Int main()

{

Struct book b1;

FILE \*fp;

fp=fopen(“hello.dat”,”wb”);

printf(“enter bookid,title,price”);

scanf(“%d”,&b1.bookid);

fflush(stdin);

gets(b1.title);

scanf(“%6.2f”,&b1.price);

fwrite(&b1,sizeof(b1),1,fp);

fclose(fp);

return 0;

}

# fread():->

* fread() function is used to read content from a file in binary mode
* int fread(void\*buffer,int size,int count,FILE\*ptr);

**Note:->** if file have a record then it return a 1 otherwise 0.

#include<stdio.h>

#include<stdlib.h>

struct book

{

Int bookid;

char title[20];

float prize;

};

Int main()

{

struct book b1;

FILE \*fp;

fp= fopen(“hello.dat”,”rb”);

if(fp==NULL)

{

printf(“cannot open”);

exit(1);

}

while(fread(&b1,sizeof(b1),1,fp)>0)

{

printf(“%d %s %6.2f”,b1.bookid,b1.title,bi.prize);

}

fclose(fp);

return 0;

}

# COMMAND LINE:->

* **way to run your programme**
  + Using IDE->like codeblock
  + Double click->.exe file
  + **Using command line**

* Command line argument is a parameter supplied to the programme.when it is invoked.
* Command line argument is an important concept of c-programming . it is mostly used when you need to control programme from outside.
* Command line Argument are passed to the main() function

Syntax:-> int main(int argc, char \*argv[])

**NOTE:->**

* **You can pass argument to the main function only when you are calling your programm using command line.**

#include<stdio.h>

Int main(int argc,char \*argv[])

{

Int i;

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

{

printf(“\n%s”,argv[i]);

}

return 0;

}

**Output**

**in background programme look like that**

|  |  |  |
| --- | --- | --- |
| 1000 | 1002 | 1004 |

main(3 , )

test1 5 6

SCREEN

Output:-> test1

C:// test1 5 6 (press enter)

5

6

**Question:-> int \*fun(); and int(\*fun)(); whats the difference?**

* int \*fun () in this fun is a function to pointer of integer type which will return pointer values of integer type
* int (\*fun)(); in this fun is a pointer to function which is returning integer values.

**int \*fun()**

Closes brackets here signify that this is a function. fun here is the name of the function. If you look at the method blueprint:

return type name (arguments)

Here modifiers and specifiers are optional. So int \* is the return type here, fun is the name of the function .

**int(\*fun)()**

Here when the \*fun is inside the parentheses, this implies a pointer to the function. Only brackets made the difference.

# Bits and Pieces:

Notes ->>>>> click on link: [Bitwise Operators in C/C++ - GeeksforGeeks](https://www.geeksforgeeks.org/bitwise-operators-in-c-cpp/)

Video: <https://youtu.be/jlQmeyce65Q>

# **Note:-> How to read a complicated deceleration**

* **Let us see the steps to read complicated declerations.**

1. Convert C declaration to postfix format and read from right to left
2. To convert Experation to postfix, start from innermost paranthesis,if innermost parenthesis is not present then start from deceleration and go right first. When first ending parenthesis encounters then go left . once whole parenthsis is parsed then come out from parenthesis
3. Continue until complete deceleration has been parsed.
4. Ex:-> int (\*fp)();
5. Let us convert above expression to postfix format
6. For the above example, there is no innermost parenthesis, that’s why , we will print decleration name i.e “fp” next step is go to right side of expression , but there is nothing on rightside of “fp to parse, that’s why go to left side. On left side we found “\*” , now print “\*” and come out of parenthesis. We will get postfix expression as below

**Postfix format:-> fp \* () int**

**Now read:-> fp is a pointer to function returning int**

**Ex:->2 int (\*daytab)[13]**

**Postfix format:->** daytab \* [13] int

**Now read:->** daytab is a pointer to array of 13 intigers.

**Ex:->3 void (\*f[10]) (int,int)**

**Postfix format:->** f [10] \* (int,int) void

**Now read:->** f is a array of 10 pointer to function which take two argument int , int and return type is void

**Ex:->4 char( \* ( \* x () )[] ) ()**

**Postfix format**:-> x () \* [] \* () char

**Now read:->** x is a function returning pointer to array of pointer to function returning char

**Ex:->5 char (\* (\* x [3] )() ) [5]**

**Postfix format:->** x [3] \* () \* [5] char

**Now read:->** x is a array of 3 pointer to function returning pointer to array of 5 character

**Ex:-> 6 int \* (\*(\* arr[5]) () ) ()**

**Postfix format:->** arr [5] \* () \* () \* int

**Now read:->** arr is a array of 5 pointer to function returning pointer to function returning pointer to int

**Ex:-> void(\*bsd\_singnal(int sig, void(\*func)(int) ) ) (int)**

**Postfix format:** bsd\_singnal (int sig, void(\*func)(int)) \* (int) void

**Now read:->** bsd\_singnal is a function that takes intiger and pointer to function (which takes argument type intiger returning void pointer to function(which take argument type int ) returning void