Software is a collection of program.

A program is a set of instructions which is designed for a particular task.

Software are classified into two types

**System software**

The software is designed for general purpose & does not contain any limitations, it is called system software.

Operating System (O.S.), language processors and utility programmes come under system software.

**Application software**

Software which is designed for a specific task is called application software. For Ex: Tally, MS OFFICE, ORACLE

**Programming Language**

It is special kind of instructions which is used to communicate with computer.

Programming language are classified into two types

1. High Level Programming language
2. Low Level Programming language

**High Level Programming language**

Programming language which is syntactically similar to English and easy to understand is called High Level Language.

By using High Level programming Lang. we can develop user interface applications.

Ex: C,C++,C#,VC++,JAVA etc.

**Low Level Programming Language**

In this programming language instructions will be there in symbolic format and that is not easy to remember all instructions along with syntax.

By using low level programming language, we are developing device driver.

Low level programming is also called Assembly language.

Ex. Microcontroller programming language (8085, 8086, 8080)

**Applications of C**

* In software field, many applications are working with the help of c-programming language only like-
* System software designing, i.e. OS and Compilers
* Application software designing i.e. databases and excel sheets.
* Any complex mathematical equations can be evaluated.
* Graphics related application, i.e. PC and mobile games.
* UNIX Kernal is completely developed in C Language.

**Introduction to C language**

C is a structured programming language developed by Dennis Ritchie in 1972 at AT&T (American Telephone & Telegraph) Bell Laboratories, USA.

C is a Middle-level programming Language, since it is used to develop applications for High Level business programs and Low Level System programs**.**

It became so popular because of its powerfulness, simplicity and easiness to use.

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

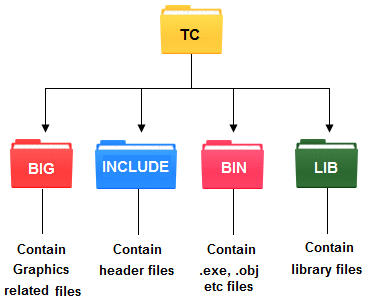
Initially, C language was developed to be used in UNIX operating system.

In the year 1988 'C' programming language standardized by ANSI (American national standard institute), that version is called ANSI-C.

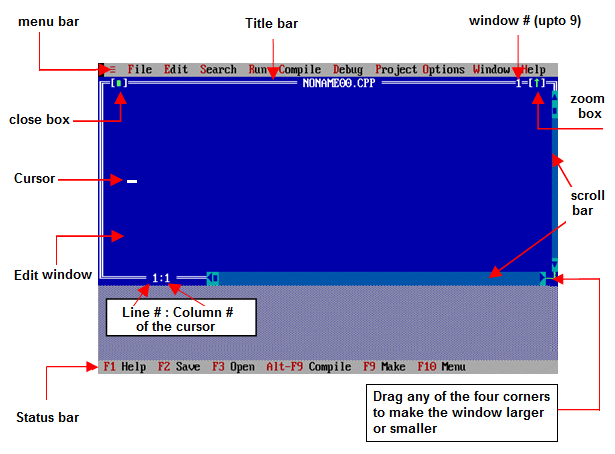
In the year of 2000 'C' programming Language standardized by 'ISO' that version is called C-99.

**Installation of TC**

When you install the Turbo C compiler on your system, then TC directory is created on the hard disk and various sub directories such as INCLUDE, and LIB etc. are created under TC.



**TC Editor**

****

**Shortcut keys Related to TC Editor**

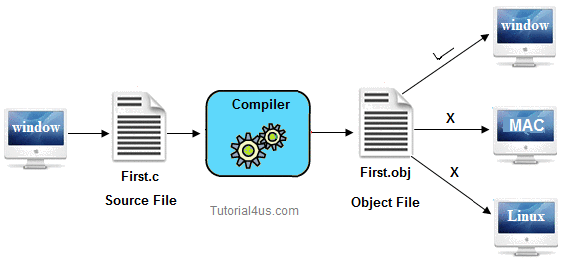
* Alt + x : Close TC Editor.
* Clt + f9 : Run C Program.
* Alt + f9 : Compile C Code.
* Alt + Enter : Get Full Screen or Half Screen TC Editor.
* Clt + y : Delete complete line above the cursor.
* Shift + Right arrow : Select Line of Code.
* Clt + Insert : Copy.
* Shift + Insert : Paste.
* Shift + Delete : Delete.

**Features of C**

* Robust language, which can be used to write any complex program.
* Has rich set of built-in functions and operators.
* Efficient and faster in execution.
* Highly portable.
* Well-suited for writing both system software and application software.
* Well-suited for structured programming.

### Platform dependent

A language is said to be platform dependent whenever the program is execute in the same operating system where that was developed and compiled but not run and execute on other operating system. C is platform dependent programming language.



### Portability

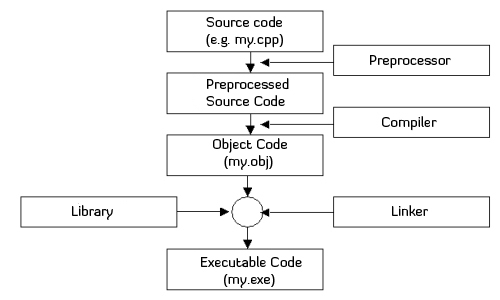
It is the concept of carrying the instruction from one system to another system. In C Language **.C** file contain source code, we can edit also this code. **.exe** file contain application, only we can execute this file. When we write and compile any C program on window operating system that program easily run on other window based system.

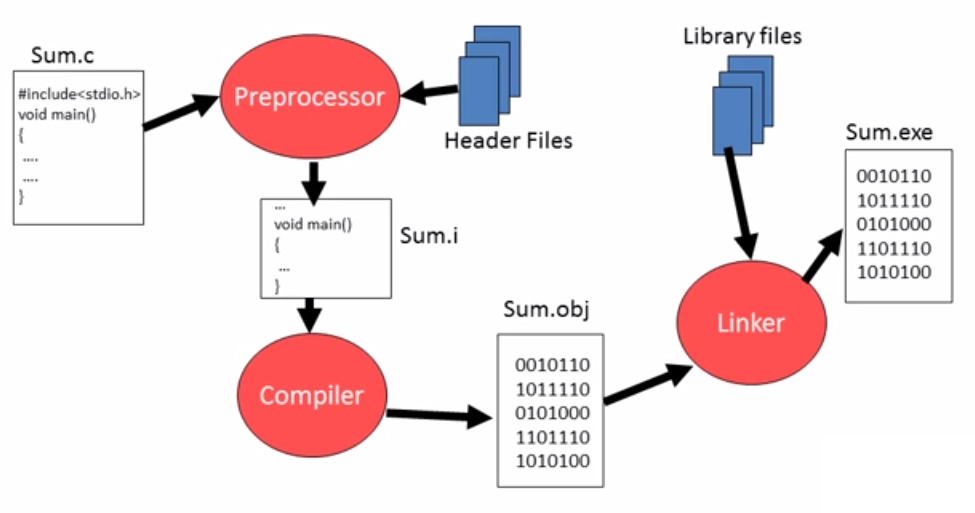
**Note:** .obj file of C program is platform dependent.

**Data Storage in a Computer**

* 4bits = 1 Nibble
* 8bits = 1 byte
* 1024 bytes = 1k or 1kb (kilobyte)
* 1024KB = 1MB (mega byte)
* 1024MB = 1GB (Gega byte)
* 1024GB = 1TB Terabytes

**Compilation Steps in C**





**Basic Structure of C program**

DOCUMENTATION SECTION.

LINK SECTION.

DEFINITION SECTION.

GLOBAL DECLARATION SECTION.

main() : FUNCTION SECTION

{

Declaration part

Executable part

}

Subprogram section

function1

function2

(User defined functions)

|

|

|

|

function n.

**First C Program**

#include <stdio.h>

#include <conio.h>

void main()

{

clrscr();

printf("Hello Affy Informatics");

getch();

}

* #include <stdio.h> includes the standard input output library functions. The printf(),scanf() function is defined in stdio.h .
* #include <conio.h> includes the console input output library functions. The clrscr(), getch() functions are defined in conio.h file.
* The main() function is the entry point of every program in c language. The void keyword specifies that it returns no value.
* The printf() function is used to print data on the console.
* The getch() function asks for a single character. Until you press any key, it hold the screen.
* clrscr() is used for clearing the output screen i.e console.

**C - Header Files**

A header file is a file with extension .h which contains C function declarations and macro definitions to be shared between several source files.

There are two types of header files: the files that the programmer writes and the files that come with your compiler.

Both user and system header files are included using the pre-processor directive #include.

It has following two forms:

**#include <filename>**

*header file is searched in standard list of system header files. This method is normally used to include standard library header files*.

**#include “filename”**

*header file is searched in the directory containing the current files.* *This method is normally used to include programmer defined header files*

**C Language – Comments**

* In the C Programming Language, you can place comments in your source code that are not executed as part of the program.
* Comments are non-executable code used to provide documentation to programmer.
* Comments provide clarity to the C source code.

**Single Line Comments**

* Single Line Comment is used to comment out just Single Line in the Code. It is used to provide One Liner Description of line.
* Single Line Comment Starts with ‘//’

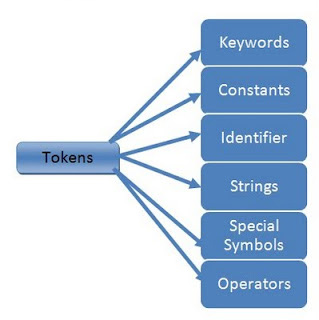
**Multi Line Comments**

* Multi line comment starts with /\*.
* Multi line comment ends with \*/.
* It can be split over multiple lines.

**C Tokens**

* Tokens are basic building blocks of C Programming.
* Each and every smallest individual unit in a C program are known as C tokens.

C tokens are of six types.



**Keywords in C**

* Keywords are reserved word.
* Keywords have special meaning in C language. The meaning has already been described. These meaning cannot be changed.
* There are total 32 keywords in C language.

|  |  |  |  |
| --- | --- | --- | --- |
| **auto** | **Double** | **int** | **struct** |
| **break** | **Else** | **long** | **switch** |
| **case** | **Enum** | **register** | **typedef** |
| **char** | **Extern** | **return** | **union** |
| **const** | **Float** | **short** | **unsigned** |
| **continue** | **For** | **signed** | **void** |
| **default** | **Goto** | **sizeof** | **volatile** |
| **Do** | **If** | **static** | **while** |

**Identifiers**

* Identifiers are user define.
* In C language identifiers are the names given to variables, constants, functions and user-define data.
* These identifier are defined against a set of rules.

**Rules for an Identifier**

* An Identifier can only have alphanumeric characters (a-z, A-Z, 0-9) and underscore ( \_ ).
* The first character of an identifier can only contain alphabet ( a-z , A-Z ) or underscore ( \_ ).
* Identifiers are also case sensitive in C. For example *name* and *Name* are two different identifier in C.
* Keywords are not allowed to be used as Identifiers.
* No special characters, such as semicolon, period, whitespaces, slash or comma are permitted to be used in or as Identifier.

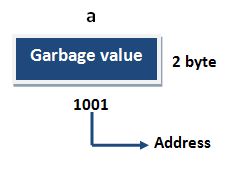
## Variable declarations

This is the process of allocating sufficient memory space for the data in term of variable.

**Syntax:**

Datatype variable\_name;

int a;



If no input values are assigned by the user than system will gives a default value called garbage value.

## Variable initialization

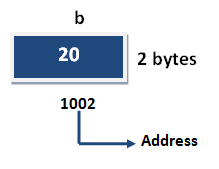
It is the process of allocating sufficient memory space with user defined values.

**Syntax:**

Datatype variable\_name = value;

Ex:

int b =20



## Variable assignment

It is a process of assigning a value to a variable.

**Syntax:**

Variable\_Name = value;

Ex:

int b;

b=25;

**Constant**

It is an identifier whose value cannot be changed at the execution time of program. In general **constant** can be used to represent as fixed values in a C program. Constants are classified into following types.

**Numeric Constant**

**Integer Constants**:

Sequence of number 0-9 without decimal points, fractional part or any other symbols. It requires two or four bytes, can be +ve, -ve or Zero. The number without a sign is assume positive. Eg: -10, +20, 40

###### Rules for Constructing Integer Constants

* An integer constant must have at least one digit.
* It must not have a decimal point.
* It could be either positive or negative.
* If no sign precedes an integer constant it is assumed to be positive.
* No commas or blanks are allowed within an integer constant.
* The allowable range for integer constants is –32768 to +32767 for 16-bit numbers. It is defined in a program as ‘int’.
* E.g. 426, +782, -8000, -7605

E.g.

|  |  |
| --- | --- |
| Invalid | Valid |
| 2\_9 | -15 |
| -13m | 2903 |
| 1.9 | 0 |

**Real Constants**

Real constants are often known as floating constants. Eg: 2.5, 5.521, 3.14 etc.

The real constants can be written in two forms, Fractional form and Exponential form.

Rules for constructing real constants expressed in Fractional form:

* A real constant must have at least one digit.
* It must have a decimal point.
* It could be either positive or negative.
* Default sign is positive.
* No commas or blanks are allowed within a real constant.
* E.g. +325.34, 426.0, -32.76, -48.5792

The exponential form of representation of real constant is usually used if the value of the constant is either too small or too large.

In exponential form of representation, the real constant is represented in two parts. The part appearing before ‘e’ is called mantissa, whereas the part following ‘e’ is called exponent.

Rules for constructing real constants expressed in exponential form:

* The mantissa part and the exponential part should be separated by a letter ‘e’.
* The mantissa part may have a positive or negative sign.
* Default sign of mantissa part is positive.
* The exponent must have at least one digit which must be a positive or negative integer. Default sign is positive.
* Range of real constants expressed in exponential form is –3.4e38 to 3.4e38.

E.g. +3.2e-5, 4.1e8, -0.2e+3, -3.2e-5

**Character Constants**

**Single character Constant**: A single character constants are given within a pair of single quote mark. Eg: 'a', '8' etc.

**String Constant**: These are the sequence of character within double quote marks

Eg: “Affy” “India”, “4”

**Declare constant**

* **const** keyword are used for declare a constant.

**Syntax**

const int height = 100;

#include<stdio.h>

#include<conio.h>

**void** main()

{

**const** **int** a=10;

printf("%d",a);

a=20; // gives error you can't modify const

getch();

}

**Operators in C Language**

C language supports a rich set of built-in operators. An operator is a symbol that tells the compiler to perform certain mathematical or logical manipulations. Operators are used in program to manipulate data and variables.

C operators can be classified into following types,

* Arithmetic operators
* Relation operators
* Logical operators
* Increment and Decrement operators
* Bitwise operators
* Assignment operators
* Ternary or Conditional operators
* Special operators

**Arithmetic operators**

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| + | Addition or unary plus |
| - | Subtraction or unary minus |
| \* | Multiplication |
| / | Division |
| % | Modulo Division ( remainder of division) |

* When both the operands in single arithmetic expression such as a+b are integers, the expression is called an integer expression, and the operation is called integer arithmetic.
* Integer arithmetic always yields an integer value.

if a and b are integers, then for a =14 b=4

a-b=10

a+b=18

a\*b=56

a/b=3(decimal part truncated)

a%b=2(remainder of division)

During modulo division, the sign of the result is always the sign of the first operand (the dividend).

-14 % 3= -2

-14 % -3= -2

14 % -3= 2

**Real Arithmetic**

An arithmetic operation involving only real operands is called real arithmetic. A real operand may assume values either in decimal or exponential notation.

6.0/7.0=0.857143

1.0/3.0=0.333333

-2.0/3.0=-0.666667

The operator % cannot be used with real operands.

**Mixed-mode Arithmetic**

When one of the operands is real and the other is integer, the expression is called a mixed-mode arithmetic expression. If either operand is of the real type, then only the real operation is performed and the result is always a real number. Thus

15/10.0=1.5

**Relational operators**

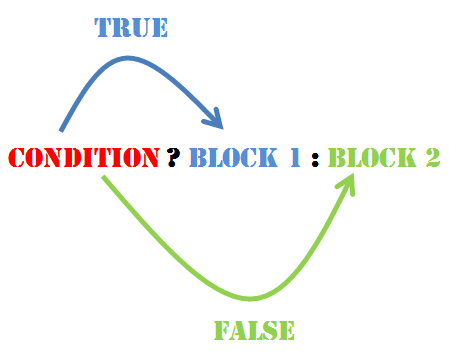
|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| == | Is equal to |
| != | Is not equal to |
| > | Is greater than |
| < | Is less than |
| >= | Is greater than or equal to |
| <= | Is smaller than or equal to |

**Logical operators**

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| && | Logical AND |
| || | Logical OR |
| ! | Logical NOT |

**Conditional / Ternary operator ?:**

* It is also known as ternary operator and used to evaluate conditional expression.
* exp1 **?** exp2 **:** exp3
* If expr1 Condition is true ? Then value expr2 : Otherwise value expr3



**Special Operator**

& - address of

sizeof();

**.** - Dot operator

, - comma operator etc.

**sizeof()**

* It returns the size of variable in bytes.
* The sizeof operator returns the size of its operand in bytes.

**Function in c**

A function is a block of code that performs a particular task.

C functions can be classified into two categories

**Library functions**

Library functions are those functions which are defined by C library, example printf(), scanf(), strcat() etc. You just need to include appropriate header files to use these functions. These are already declared and defined in C libraries.

**User-defined functions**

User-defined functions are those functions which are defined by the user at the time of writing program.

* Functions are made for code reusability and for saving time and space.

**Benefits of Using Functions**

* It provides modularity to the program. We can divide c program in smaller modules. Modular programming makes c program more readable.
* Easy code Reusability. You just have to call the function by its name to use it.
* In case of large programs with thousands of code lines, debugging and editing becomes easier if you use functions.
* Code optimization-It makes the code optimized, we don't need to write much code.

**Function declaration/Prototype**

General syntax of function declaration is,

***return-type* function-name (parameter-list) ;**

**Function definition**

General syntax of function definition is,

***return-type* function-name (parameter-list)**

**{**

**local variable declaration;**

**statements;**

**}**

* The first line *return-type* function-name(parameter) is known as function header and the statement within curly braces is called function body.

**Function Call**

function-name(argument-list);

**Function arguments**

The function parameters are the means of communication between the calling and the called functions. There is no limitation on the number of parameters passed to function.

**Formal parameters:**

These, commonly called as parameters, are given in the function declaration and function definition.

**Actual parameters:**

These, commonly called as arguments, are specified in the function call.

**The following conditions must be satisfied for function call:**

* The list of arguments in the function call and function declaration must be the same.
* The data type of each of the actual parameter must be same as that of formal parameter.
* The order of the actual parameters must be same as the order in which the formal parameters are specified.

## Difference between Local variable and Global variable

In C language, a variable can be either of global or local scope.

## Global variable

Global variables are defined outside of all the functions, generally on top of the program. The global variables will hold their value throughout the life-time of your program.

## Local variable

A local variable is declared within the body of a function or a block. Local variable only use within the function or block where it is declare.

**Example of Global and Local variable**

Example

#include<stdio.h>

#include<conio.h>

int a; // global variable

void main()

{

int b; // local variable

a=10, b=20;

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

printf("Value of b : %d",b);

getch();

}

Output

Value of a: 10

Value of b: 20

**Recursion**

When Function is call within same function is called **Recursion**. The function which calls same function is called **recursive function**. In other word when a function calls itself then that function is called **Recursive function**.

#include<stdio.h>

#include<conio.h>

int factorial(int n)

{

if(n==0)

return(1);

else

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

}

void main( )

{

int result;

result=factorial(5);

printf(“%d”,result);

}

**Array**

* In C language,  An array is defined as finite ordered collection of homogenous (same) type, stored in contiguous memory locations.
* Subscript references individual Elements.
* Subscript also called index.
* In C, index start at 0 to (n-1). cannot be Negative. where n is size of array.

**Type of Arrays**

* One-dimensional arrays
* Two-dimensional arrays
* Multi-dimensional arrays

**Declaring One-Dimensional Array**

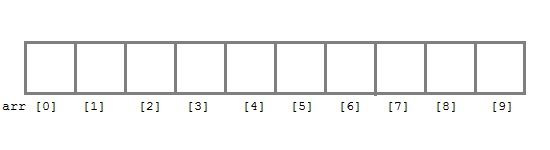
Like any other variable, arrays must be declared before they are used. General form of array declaration is,

**data-type array-name[size];**

* data-type specifies the type of element that will be contained in the array.
* Size indicates the maximum number of elements that can be stored inside the array.

for example :

int arr[10];



**Initialization of an Array**

After an array is declared it must be initialized. Otherwise, it will contain garbage value (any random value). An array can be initialized at either compile time or at runtime.

**Compile time Array initialization**

*type* array-name[size] = { list of values };

int marks[4]={ 67, 87, 56, 77 };

float area[5]={ 23.4, 6.8, 5.5 };

One important thing to remember is that when you will give more initializer than declared array size than the compiler will give an error.

If the number of values in the list is less than the number of elements, then only that many elements will be initialized. The remaining elements will be set to zero automatically.

The size may be omitted. In such cases, the compiler allocates enough space for all initialized elements.

int arr[ ]={10,20,30,40,50};

**Runtime Array initialization**

An array can also be initialized at runtime using scanf() function. This approach is usually used for initializing large array, or to initialize array with user specified values.

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();

}

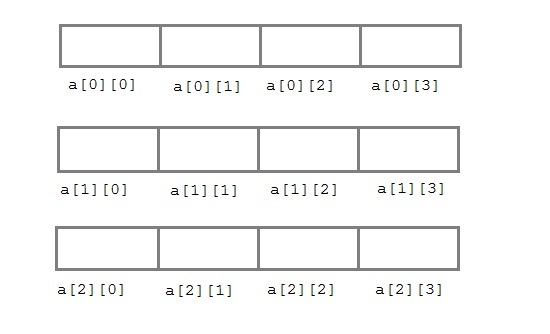
**Two dimensional Arrays**

C language supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.

Two-dimensional array is declared as follows,

***data-type array-name[row-size][column-size];***

Example : int a[3][4];



**Compile time initialization**

int a[3][4]={

{10 , 20 , 30 , 40},

{50, 60 , 70 , 80},

{50, 30 , 21 , 45}

};

when the array is completely initialized with all values explicitly , we need not specify the size of the first dimension.

int a[ ][4]={

{10 , 20 , 30 , 40},

{50, 60 , 70 , 80},

{50, 30 , 21 , 45}

};

OR

int a[3][4]={10 , 20 , 30 , 40, 50, 60 , 70 , 80, 50, 30 , 21 , 45};

**Runtime Initialization**

void main()

{

int arr[3][4];

int i,j,k;

printf("Enter array element");

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

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

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

}

}

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

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

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

}

}

}

**String**

* string is a sequence of characters that is treated as a single data item and terminated by null character '\0'.
* C language does not support string as a data type.
* A string is actually one-dimensional array of characters in C language.
* For example: The string "hello world" contains 12 characters including '\0' character which is automatically added by the compiler at the end of the string.

**Declaring and Initializing a string variables**

The general form of declaration of a string variable is:

**char string\_name[size];**

**Ex:**

**char city[10];**

**char name[30];**

When the compiler assign a character string to a character array, it automatically supplies a null character (‘\0’) at the end of the string. Therefore the size should be equal to the maximum number of characters in the string *plus* one.

There are different ways to initialize a character array variable.

char name[5]=“Affy";

char name[5]={'A',‘f','f',‘y','\0'};

Remember that when you initialize a character array by listings all its characters separately then you must supply the '\0' character explicitly.

String can also be declare and Initialized using pointer

**char \*ch=“Affy";**

**Reading Strings from user**

char name[20];

scanf("%s",name);

**gets() and puts()**

Functions gets() and puts() are two string functions to take string input from user and display string respectively.

#include<stdio.h>

int main()

{

char name[30];

printf("Enter name: ");

gets(name); //Function to read string from user.

printf("Name: ");

puts(name); //Function to display string.

return 0;

}

**Note:** Though, gets() and puts() function handle string, both these functions are defined in "stdio.h“ header file.

**Built-in String Functions**

The header file string.h provides useful set of string functions. These functions help in manipulating strings.

To use these functions, the header file string.h must be included in the program with the statement:

**#include<string.h>**

**Built-in String Functions**

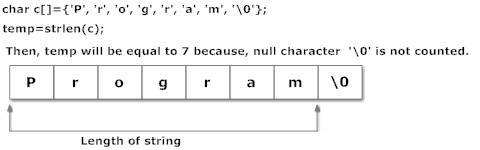
|  |  |
| --- | --- |
| **Method** | **Description** |
| strcat() | It is used to concatenate(combine) two string |
| strlwr() | Converts a string to lowercase. |
| strupr() | Converts a string to uppercase. |
| strlen() | It is used to show length of a string |
| strrev() | It is used to show reverse of a string |
| strcpy() | Copies one string into another |
| strcmp() | It is used to compare two string |

**strlen()**

* In C, strlen() function calculates the length of string. It takes only one argument, i.e, string name.

**strlen(string\_name);**

**Ex:**



#include <stdio.h>

#include <string.h>

int main()

{

char a[20]="Program";

char b[20]={'P','r','o','g','r','a','m','\0'};

char c[20];

printf("Enter string: ");

gets(c);

printf("Length of string a=%d \n",strlen(a));

printf("Length of string b=%d \n",strlen(b));

printf("Length of string c=%d \n",strlen(c));

return 0;

}

**Output:**

Enter string: Affy

Length of string a=7

Length of string b=7

Length of string c=4

**strcat()**

In C programming, strcat() concatenates(joins) two strings. It takes two arguments, i.e, two strings and resultant string is stored in the first string specified in the argument.

**strcat(first\_string, second\_string);**

#include <stdio.h>

#include <string.h>

int main()

{

char str1[]=“Affy ", char str2[]=“Informatics";

strcat(str1,str2); //concatenates str1 and str2 and resultant string is stored in str1.

puts(str1);

puts(str2);

return 0;

}

Output:

AffyInformatics

Informatics

**strcpy()**

Function strcpy() copies the content of one string to the content of another string. It takes two arguments.

**strcpy(destination,source);**

Here, source and destination are both the name of the string. This statement, copies the content of string source to the content of string destination.

#include <stdio.h>

#include <string.h>

int main()

{

char a[10],b[10];

printf("Enter string: ");

gets(a);

strcpy(b,a); //Content of string a is copied to string b.

printf("Copied string: ");

puts(b);

return 0;

}

Output:

Enter string: Affy

Copied string: Affy

**strupr()**

**strupr(string\_name);**

**strlwr()**

**strlwr(string\_name);**

#include <stdio.h>

#include <string.h>

int main()

{

char str1[]=“AffyInformatics";

puts(strupr(str1)); //Converts to uppercase and displays it.

puts(strlwr(str1)); //converts to lowercase and displays it.

return 0;

}

**strcmp()**

The function strcmp() is used to compare two strings.

The function accepts two strings as parameters and returns an integer value, depending upon the relative order of two strings.

* Returns 0 if str1 is same as str2.
* Returns <0 if strl is less than str2.
* Returns >0 if str1 is greater than str2

#include <stdio.h>

#include <string.h>

int main()

{

char str1[30],str2[30];

printf("Enter first string: ");

gets(str1);

printf("Enter second string: ");

gets(str2);

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

printf("Both strings are equal");

else

printf("Strings are unequal");

return 0;

}

**strrev()**

#include<stdio.h>

#include<string.h>

int main()

{

char name[30] = "Hello";

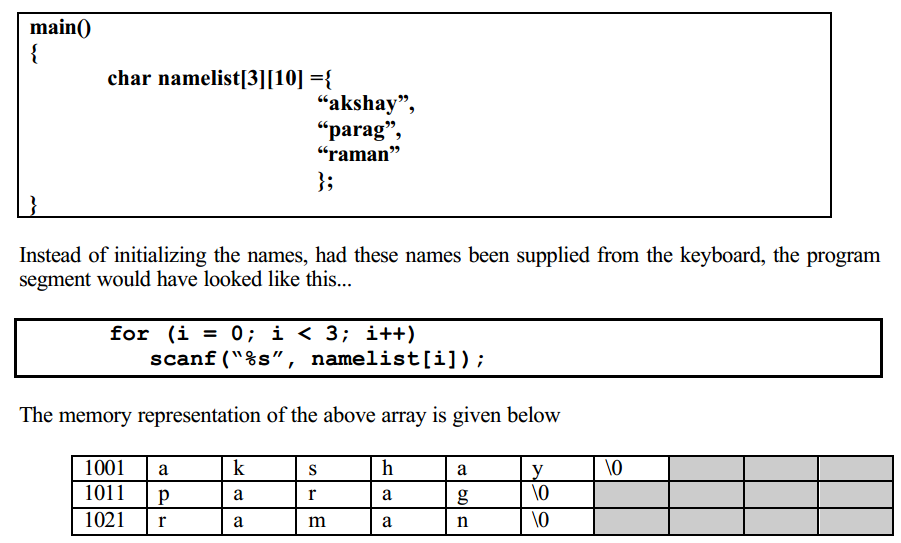
printf("String before strrev( ) : %s\n",name);

printf("String after strrev( ) : %s",strrev(name));

return 0;

}

**Two Dimensional Arrays of Characters**



**Buffer**

* Temporary storage area is called buffer.
* All standard input output devices are containing input output buffer.
* In implementation when we are passing more than required number of values as a input then rest of all values will automatically holds in standard input buffer, this buffer data will automatically pass to next input functionality if it is exist.

#include<stdio.h>

#include<conio.h>

**void** main()

{

**int** v1,v2;

clrscr();

printf("\n Enter v1 value: ");

scanf("%d",&v1);

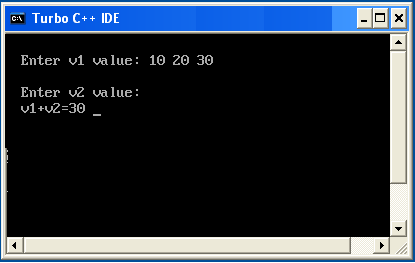
printf("\n Enter v2 value: ");

scanf("%d",&v2);

printf("\n v1+v2=%d ",v1+v2);

getch();

}



In implementation when we need to remove standard input buffer data then go for flushall() or fflush() function.

**flushall()**

it is a predefined function which is declared in stdio.h. by using flushall we can remove the data from standard input output buffer.

**fflush()**

it is a predefined function in "stdio.h" header file used to flush or clear either input or output buffer memory.

**fflush(stdin)**

it is used to clear the input buffer memory. It is recommended to use before writing scanf statement.

**fflush(stdout)**

it is used to clear the output buffer memory. It is recommended to use before printf statement.

#include<stdio.h>

#include<conio.h>

**void** main()

{

**int** v1,v2;

clrscr();

printf("\n Enter v1 value: ");

scanf("%d",&v1);

printf("\n Enter v2 value: ");

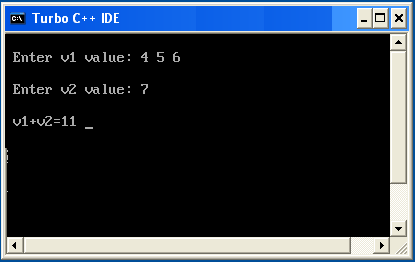
flushall(); //fflush(stdin);

scanf("%d",&v2);

printf("\n v1+v2=%d ",v1+v2);

getch();

}



**Command Line Arguments**

Parameters or Values can be passed to program from the command line which are received and processed in main function. Since the arguments are passed from the command line hence they are called as command line arguments. All commands on Unix Operating System use this concept.

Two built in formal parameters are used to accept parameters in main.

* **argc** : Contains number of command line arguments. It is type int.
* **argv :** A pointer to an array of strings where each string represents a token of the arguments passed. It is a character array of pointers.
* **main(int argc, char \*argv[])**

#include <stdio.h>

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

{

if( argc == 2 )

{

printf("The argument supplied is %s\n", argv[1]);

}

else if( argc > 2 )

{

printf("Too many arguments supplied.\n");

}

else

{

printf("One argument expected.\n");

}

}

**C Program to Add two numbers using Command Line Arguments**

#include<stdio.h>

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

{

   int i, sum = 0;

   if (argc != 3)

{

      printf("You have forgot to type numbers.");

      exit(1);

   }

   printf("The sum is : ");

   for (i = 1; i < argc; i++)

      sum = sum + atoi(argv[i]);

   printf("%d", sum);

}

**Structure**

* Structure is a user-defined data type in C which allows you to combine different data types.
* It is somewhat similar to an Array. The only difference is that array is used to store collection of similar data types while structure can store collection of any type of data.
* Structure is used to represent a record. Suppose you want to store record of Student which consists of student name, address, roll number and age. You can define a structure to hold this information.

**Defining a structure**

* **struct** keyword is used to define a structure. **struct** define a new data type which is a collection of different type of data.

**Syntax:**

**struct** structure\_name

{

     data\_type  member1;

     data\_type  member2;

     .

    .

     data\_type  memeberN;

};

Ex:

struct Book

{

char title[20];

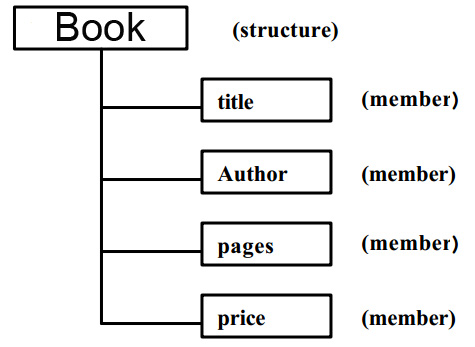
char Author[20];

int pages;

float price;

};

**Structure of Book**



* All the members of a structure can be of same type.

struct date

{

int day,month,year;

} ;

**Declaring Structure Variables**

It is possible to declare variables of a **structure**, after the structure is defined. **Structure** variable declaration is similar to the declaration of variables of any other data types. Structure variables can be declared in following two ways.

* **Declaring Structure variables separately**
* **Declaring Structure Variables with Structure definition**

**Declaring Structure variables separately**

struct Student

{

char name[20];

int age;

int rollno;

} ;

//Now write given code inside the main() function.

void main()

**{**

**struct Student S1 , S2;** //declaring variables of Student

}

**Declaring Structure Variables with Structure definition**

struct Student

{

char name[20];

int age;

int rollno;

} **S1**, **S2** ;

Here **S1** and **S2** are variables of structure **Student**.

**Accessing Structure Members**

* To access any member of a structure, we use the member access operator (.).
* (.) operator also called period operator.
* Structure-variable . member-name

**Structure Initialization**

struct Student

{

char name[20];

int age;

int rollno;

} S1={“dheeraj”,21,101};

OR

Struct Student **S1 =** {“dheeraj”,21,101};

OR

Struct Student **S1;**

S1.name=“dheeraj”; **//initialization of each member separately**

S1.age=21;

S2.rollno=101;

#include<stdio.h>

#include<conio.h>

struct book

{

char title[20];

char author[20];

int pages;

};

void main()

{

struct book b1={“C”,”Anuj”,100};

clrscr();

printf("title=%s author=%s page=%d",b1.title,b1.author,b1.pages);

getch();

}

#include<stdio.h>

#include<conio.h>

struct book

{

char title[20];

char author[20];

int pages;

};

void main()

{

struct book b1;

clrscr();

printf("Enter the title,author,pageno\n");

scanf("%s %s %d",b1.title,b1.author,&b1.pages);

printf("title=%s author=%s page=%d",b1.title,b1.author,b1.pages);

getch();

}

**Nested Structure**

Nesting of structures is also permitted in C language.

**The First way**

struct date

{

int day,month,year;

};

struct emp

{

char name[15];

**struct date birthday;**

int salary;

};

**The Second way**

struct emp

{

char name[15];

**struct date**

**{**

**int day,month,year;**

**}birthday;**

int salary;

};

#include<stdio.h>

#include<conio.h>

struct date

{

int day,month,year;

};

struct emp

{

char name[20];

**struct date d;**

};

int main()

{

struct emp e;

clrscr();

printf(“Enter name , day , month , year\n”);

scanf("%[^\n]",e.name);

scanf("%d %d %d",&e.d.day,&e.d.month,&e.d.year);

printf("%s %d / %d / %d",e.name,e.d.day,e.d.month,e.d.year);

getch();

return 0;

}

**Structure and Array**

* Array of Structures
* Structures Containing Arrays

**Array of Structures**

In the array of structure array contains individual structures as its elements. These are commonly used when a large number of similar records are required to be processed together.

**struct Student s[10];**

This statement declares s to be an array containing 10 elements of type Student.

#include<stdio.h>

#include<conio.h>

struct book

{

int pageno;

char title[20];

char author[20];

};

void main()

{

struct book b[5];

int i;

clrscr();

printf("Enter the book Information");

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

{

scanf("%d %s %s",&b[i].pageno,b[i].title,b[i].author);

}

printf("The book inf\n");

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

{

printf("%d %s %s",b[i].pageno,b[i].title,b[i].author);

}

getch();

}

**Structures and Pointer**

* Pointers to Structures
* Structures Containing Pointer

**Pointer to structure**

* **Pointer Variable** which stores the **address of Structure** must be declared as **pointer to structure** .
* You can define a pointer to a structure in the same way as any pointer to any type.

struct student

{

char name[10];

int roll;

int marks;

}stud1;

struct student \*ptr;

ptr = &stud1;

* The pointer variable *ptr* can now be used to access the member variable of *student* using the dot operator as

(\*ptr).name;

(\*ptr).roll;

(\*ptr).marks;

* The parentheses are necessary because the dot operator (.) has higher precedence than that of the dereferencing operator (\*).
* The members can also be accessed by using arrow operator ( ->)*.*

Ex: **Pointer to Structure**

#include<stdio.h>

struct city

{

int dist;

char cityname[20];

}b,\*ptr;

void main()

{

ptr=&b;

printf("Enter the city distance and cityname info\n");

scanf("%d %s",&ptr->dist, ptr->cityname);

printf("%d %s\n",ptr->dist, ptr->cityname);

printf(“%d %s\n”,(\*ptr).dist, (\*ptr).cityname);

}

**Structure Containing Pointers**

* A structure can contain pointers as member variables.
* Pointers are used to store the address of memory location.
* They can be **de-referenced** by **‘\*’** operator.

struct Location

{

char \*name;

char \*address;

};

**Example: pointer within structure**

#include<stdio.h>

struct Student

{

int \*ptr; //Stores address of integer Variable

char \*name; //Stores address of Character String

}s1;

int main()

{

int roll = 20;

s1.ptr = &roll;

s1.name = “Anuj";

printf(“\nRoll Number of Student : %d",\*s1.ptr);

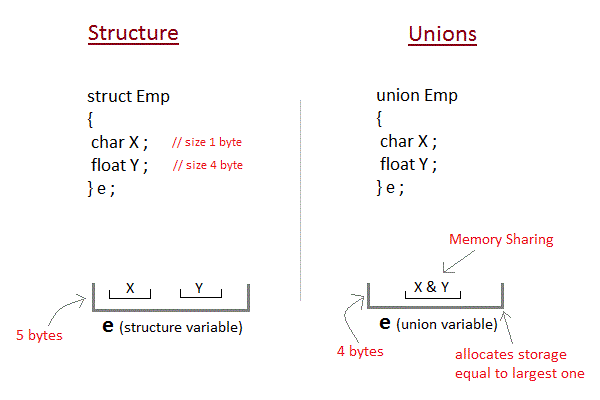
printf("\nName of Student : %s",s1.name);

return(0);

}

**Unions in C Language**

* Unions are conceptually similar to structures.
* The syntax of union is also similar to that of structure.
* The only difference is in terms of storage. In structure each member has its own storage location, whereas all members of union use a single shared memory location which is equal to the size of its largest data member.



**Defining Union**

The **union** keyword is used to define union. Let's see the syntax to define union in c.

**union** union\_name

{

    data\_type member1;

    data\_type member2;

    .

    .

    data\_type memeberN;

};

**Advantage of Union over Structure**

* It **occupies less memory** because it occupies the memory of largest member only.

Example sizeof structure v/s union

#include<stdio.h>

struct A {

char x[10];

int y;

float z;

};

union B {

char x[10];

int y;

float z;

};

void main()

{

printf("size of struct A is %d\n",sizeof(struct A));

printf("size of union B is %d",sizeof(union B));

}

**Output:**

size of struct A is 16

size of union B is 10

**Preprocessor in C**

* Preprocessor is a program which will executed automatically before passing the source program to compiler. This process is called pre-processing.
* Processing is under the control of preprocessor directives.
* Commands used in preprocessor are called preprocessor directives and they begin with pond "#" symbol and should be not ended with (;).
* When we are working with preprocessor directives, it can be placed anywhere within the application, but generally recommended to place at the top of the program.

In C-Programming language, preprocessor directives are classified into 4 types-

1) Macro Substitution directives

#define

2) File inclusion directives

#include

3) Conditional Compilation directives

#if, #else, #endif, #elif, #ifdef, #ifndef, #undef

4) Miscellaneous directive

#pragma, #error, #line

**Macro Substitution**

#define identifier token\_string

**C Program to find area of a cricle.**

#include <stdio.h>

**#define PI 3.14**

int main()

{

int radius;

float area;

printf("Enter the radius: ");

scanf("%d",&radius);

area=PI\*radius\*radius;

printf("Area=%.2f",area);

return 0;

}

#define size 100

void main()

{

int x;

x=++size; //x=++100;

printf(“x=%d”,x);

}

Error : L value required

By using #define, we can create symbolic constant values i.e. can’t be changed at time of execution.

In the above program size is an identifier which is replaced with 100 at the time of preprocessing , so it is not possible to change the value.

#define A 2+3

#define B 4+5

void main()

{

int C;

C=A\*B; // C=2+3\*4+5

printf(“C=%d”,C); // 19

}

#define A (2+3)

#define B (4+5)

When we write C=(A)\*(B)

= (2+3)\*(4+5)

=5\*9 =45

**#define pf printf**

**#define sc scanf**

#define start main()

void start

{

int a,b;

pf(“Enter two values”);

sc(“%d %d”,&a,&b);

pf(“Sum=%d”,a+b);

}

#define max(a,b) a>b?a:b

void main() {

int m;

m=max(10,20); //m=10>20?10:20

printf(“Max value = %d”,m);

}

#define sum(a,b) a+b

void main() {

int s;

s=sum(10,20); //s=10+20

printf(“Sum= %d”,s);

}

#define SQR(a) a\*a

void main()

{

int i,j;

i=SQR(2); //2\*2

j=SQR(2+3); //2+3\*2+3

printf(“i= %d j=%d”,i,j);

}

OUTPUT

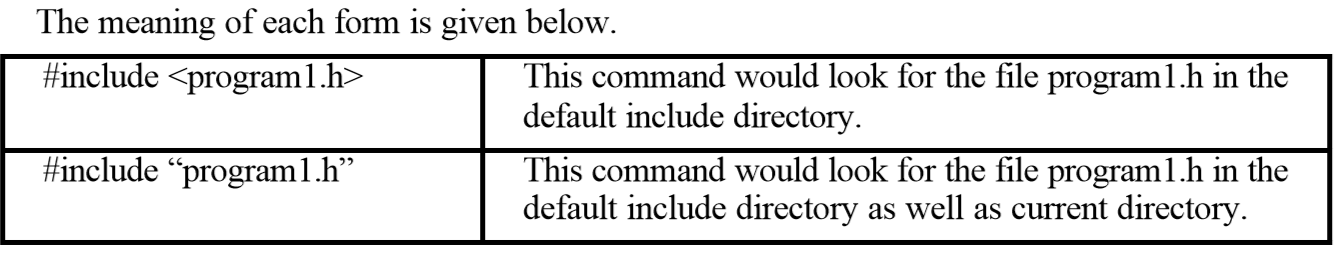
i=4 j=11

**File Inclusion**

By using this preprocessor we can include a file into another file.

#include<filename>

#include “filename”



**Typedef**

The C programming language provides a keyword called **typedef**, by using this keyword you can create a user defined name for existing data type. Generally typedef are use to create an **alias name** (nickname).

**Declaration of typedef**

typedef datatype alias\_name;

Ex:

**typedef** **int** Intdata;

#include<stdio.h>

#include<conio.h>

typedef int Intdata; // Intdata is alias name of int

void main()

{

int a=10;

Integerdata b=20;

typedef Intdata Integerdata; // Integerdata is again alias name of Intdata

Integerdata s;

clrscr();

s=a+b;

printf("\n Sum:= %d",s);

getch();

}

OUTPUT:

Sum:=20

**Pointer**

* Pointer is a variable that hold address of another variable.
* Pointer always consumes 2 bytes in memory.

**Address of Operator**

& is known as address of operator.

It is a Unary Operator.

Operand must be the name of variable.

& operator gives address number of variable.

& is also known as referencing operator.

#include<stdio.h>

void main()

{

int a=50;

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

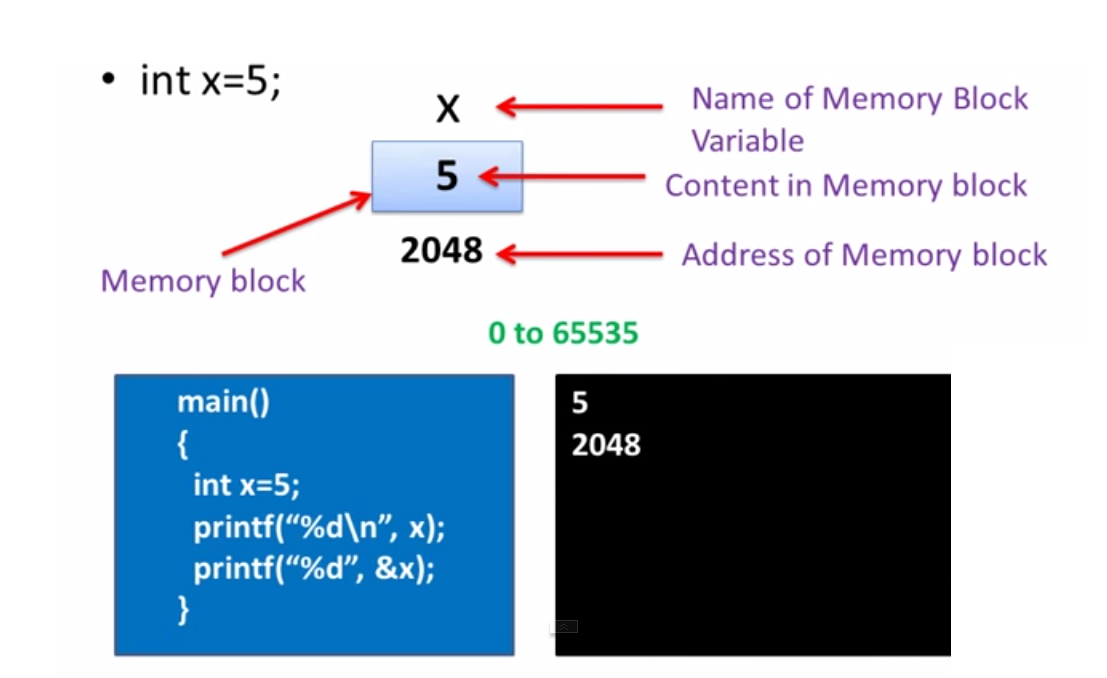
printf("\nAddress of a is: %u",&a);

}

Output:

Value of a is: 50

Address of a is: 1002



**Indirection Operator**

\* is indirection operator.

It is also known as **dereferencing operator**.

It is unary operator.

It takes address as an argument.

\* returns the content / container whose address is its argument.

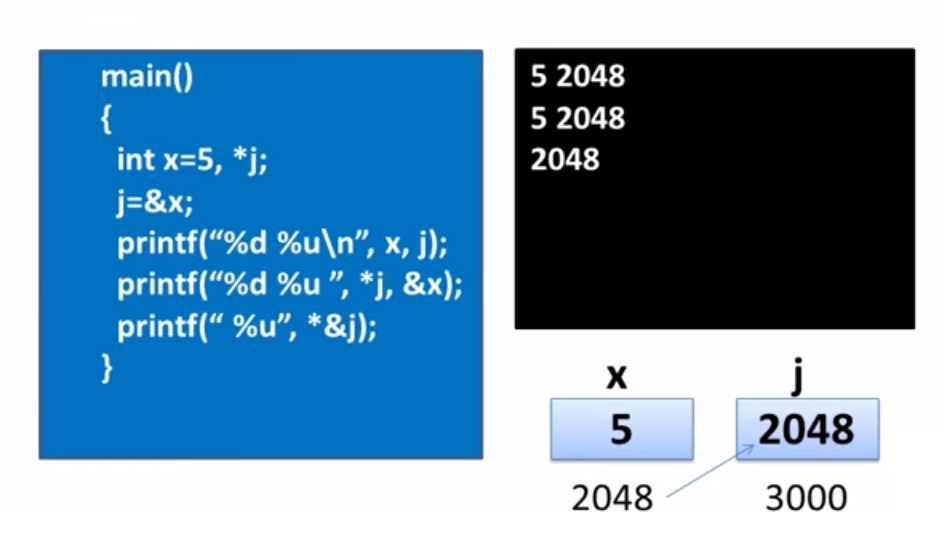
**Pointer Declaration**

In C language for declared pointer we can use \* (asterisk symbol).

**datatype \*pointervariable;**

**Pointer Assignment**

**pointervariable = &variable;**

****

**Pointer Rules**

* Pointer Variable can be assigned the address of another Variable.
* Pointer Variable can be Assigned the value of another Pointer Variable
* Pointer Variable can be initialized with zero or NULL value.
* Pointer variable can be Perform Pre/Post fix Increment/Decrement Operation.
* Integer value can be added or Subtracted from Pointer variable.
* When two Pointer points to same array then one Pointer variable can be subtracted from another.
* When Two Pointers pointing to objects of same data type then they can be compared using the Relational Operator.
* Value cannot be assigned to arbitrary address.
* Pointer Variable cannot be multiplied by Constant.
* Two pointer variable cannot be added, multiply or divide.
* We cannot multiply an integer to Pointer variable.
* We cannot Divide pointer variable with an integer value.

**NULL Pointers in C**

* 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.
* The NULL pointer is a constant with a value of zero defined in several standard libraries.

#include <stdio.h>

int main ()

{

int \*ptr = NULL;

printf("The value of ptr is : %d\n", ptr );

return 0;

}

Output:

The value of ptr is 0

**Pointers and Arrays**

* When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.
* The base address is the location of first element (index 0) of the array.
* The compiler also defines the array name as constant pointer to the first element. Suppose we declare an array x as follows:

int x[5] = {10,20,30,40,50};

* The name x is defined as a constant pointer pointing to the first element, x[0] and therefore the value of x is 1000, the location where x[0] is stored. That is

x=&x[0]=1000

* If we declare p as an integer pointer, then we can make the pointer p to point to the array x by the following assignment:

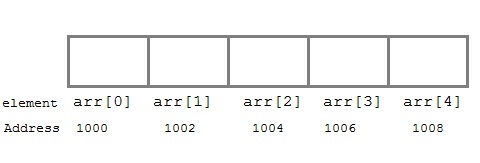
p=x;

This is equivalent to

p=&x[0];

Now we can access every value of x using p++ to move from one element to another.

* p=&x[0] (= 1000)
* p+1=&x[1] (= 1002)
* p+2=&x[2] (= 1004)
* p+3=&x[3] (= 1006)
* p+4=&x[4] (= 1008)



NOTE: You cannot decrement a pointer once incremented. p-- won't work.

**Pointer to Array**

#include<stdio.h>

void main()

{

int i;

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

int \*p = a; // same as int \*p = &a[0]

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

{

printf("%d", \*p);

p++;

}

}

**Pointers and Multidimensional Arrays**

**File Handling**

A file represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data.

C supports a number of functions that have the ability to perform basic file operations which include:

* Naming a file
* Opening a file
* Reading data from file
* Writing data to a file
* Closing a file

**Function description**

fopen() create a new file or open a existing file

fclose() closes a file

getc() reads a character from a file

putc() writes a character to a file

fscanf() reads a set of data from a file

fprintf() writes a set of data to a file

getw() reads a integer from a file

putw() writes a integer to a file

fseek() set the position to desire point

ftell() gives current position in the file

rewind() set the position to the beginning point

**Defining and Opening a File**

If we want to store data in file in the secondary memory, we must specify certain things about the file, to the operating system. They include:

1. Filename
2. Data structure
3. Purpose

Data structure of a file is defined as FILE in library of standard I/O function definition.

Therefore, all files should be declared as type FILE before they are used. FILE is a defined data type.

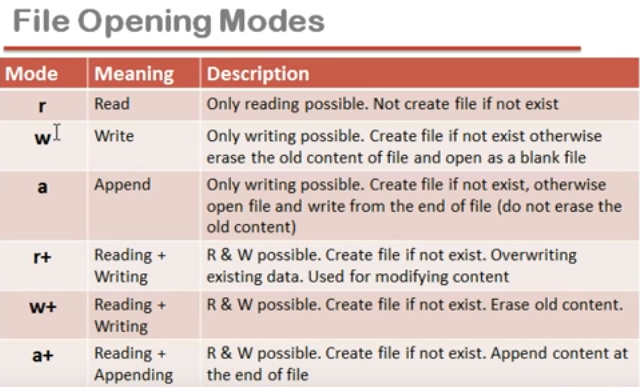
When we open a file, we must specify what we want to do with the file. For example we may write the data to the file or read the already existing data.

Following is the general format for declaring and opening a file:

FILE \*fp;

fp = fopen ( “ filename ”, ”mode ” );

Here fp is the FILE pointer, which will hold the reference to the opened (or created) file.



**Closing a File**

The fclose() function is used to close an already opened file.

General Syntax :

int fclose( FILE \*fp );

Here fclose() function closes the file and returns zero on success, or EOF if there is an error in closing the file. This EOF is a constant defined in the header file stdio.h.

…...

FILE \*p1,\*p2;

p1 = fopen(“INPUT”,”w”);

p2 = fopen(“OUTPUT”,”r”);

……

…...

fclose(p1);

fclose(p2);

……

**Input/Output operation on File**

In the above table we have discussed about various file I/O functions to perform reading and writing on file. getc() and putc() are simplest functions used to read and write individual characters to a file.

#include<stdio.h>

#include<conio.h>

main()

{

FILE \*fp;

char ch;

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

printf("Enter data");

while( (ch = getchar()) != EOF)

{

putc(ch,fp);

}

fclose(fp);

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

while( (ch = getc(fp)! = EOF)

printf("%c",ch);

fclose(fp);

}

**Difference between Append and Write Mode**

Write (w) mode and Append (a) mode, while opening a file are almost the same. Both are used to write in a file. In both the modes, new file is created if it doesn't exists already.

The only difference they have is, when you open a file in the write mode, the file is reset, resulting in deletion of any data already present in the file. While in append mode this will not happen. Append mode is used to append or add data to the existing data of file(if any). Hence, when you open a file in Append(a) mode, the cursor is positioned at the end of the present data in the file.

**Dynamic Memory Allocation**

The process of allocating memory at run time is called dynamic memory allocation.

C language offers 4 dynamic memory allocation functions. They are,

* malloc()
* calloc()
* realloc()
* free()

**malloc**

Allocates requested size of bytes and returns a pointer to the first byte of the allocated space.

By using malloc() we can create the memory dynamically at initial stage. Malloc() required one argument of type size type that is data type size malloc() will creates the memory in bytes format. Malloc() through created memory initial value is garbage.

void \* malloc(byte-size);

Note: Dynamic memory allocation related function can be applied for any data type that's why dynamic memory allocation related functions return void\*.

When we are working with dynamic memory allocation type specification will be available at the time of execution that's why we required to use type casting process.

int \*ptr;

ptr=(int \*)malloc(sizeof (int)); //2 byte

char \*cptr;

cptr=(char \*)malloc(sizeof(char)); //1 byte

int \*arr;

arr=(int \*)malloc(sizeof(int)\*10); //20 byte

cahr \*str;

str=(char \*)malloc(sizeof(char)\*50); //50 byte

**calloc**  
Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.

* By using calloc() we can create the memory dynamically at initial stage.
* calloc() required 2 arguments of type count, size-type.
* count will provide number of elements; size-type is data type size.
* calloc() will creates the memory in blocks format.
* Initial value of the memory is zero.

int \*arr;

arr=(int \*)calloc(10, sizeof(int)); // 20 byte

char \*str;

str=(char \*)calloc(50, sizeof(char)); // 50 byte

**realloc**  
Modifies the size of previously allocated space.

* By using realloc() we can create the memory dynamically at middle stage.
* Generally by using realloc() we can reallocation the memory.
* Realloc() required 2 arguments of type void\*, size\_type.
* void\* will indicates previous block base address, size-type is data type size.
* Realloc() will creates the memory in bytes format and initial value is garbage.

void\* realloc(void\*, size-type);

int \*arr;

arr=(int\*)calloc(5, sizeof(int));

........

arr=(int\*)realloc(arr,sizeof(int)\*10);

**free**  
Frees previously allocated space.

* When we are working with dynamic memory allocation memory will created in heap area of data segment.
* When we are working with dynamic memory allocation related memory it is a permanent memory if we are not de-allocated that's why when we are working with dynamic memory allocation related program, then always recommended to deleted the memory at the end of the program.
* By using free() dynamic allocation memory can be de-allocated.
* free() requires one arguments of type void\*.

**void free(void \*);**

int \*arr;

arr=(int\*)calloc(10,sizeof(int));

...

......

free(arr);

#include<stdio.h>

#include<conio.h>

#include<alloc.h>

#define NULL 0

void main()

{

int \*ptr;

int i,n,sum=0;

float avg;

clrscr();

printf("enter the number of element you want to store in the array");

scanf("%d",&n);

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

if(ptr==NULL)

{

printf(" memory not available");

getch();

exit(0);

}

else

{

printf("Enter the val");

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

{

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

}

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

{

sum=sum+(\*(ptr+i));

}

printf("sum of %d elements of array is=%d",n,sum);

avg=sum/n;

printf("The avg=%f ",avg);

}

getch();

}

//program illustrates the use of function free() and realloc()

#include<stdio.h>

#include<conio.h>

#include<alloc.h>

#include<string.h>

void main()

{

char \*msg;

clrscr();

msg=(char \*)malloc(30\*sizeof(char));

strcpy(msg,"all is well");

printf("The message now is %s\n",msg);

msg=(char \*)realloc(msg,50);

strcpy(msg,"i m fine....");

printf("\nThe message is now %s",msg);

free(msg);

getch();

}

**Graphics:**

* Applying the visual properties to a dos application is called Graphics.
* Whenever we work with graphics application, always we require including **"graphics.h"**
* Whenever we work with graphics application then we require finding out "EGAVGA.BGI" file location. It will install all the resources of graphics to the application.
* Generally the file is available in "C:\TC\BGI" directory.
* Whenever we work with graphics application then we require initializing graphics properties properly.
* By using "initgraph()" function we can initialize graphics property.
* At the time of initializing the graphics due to graphics resource we can get initialization related errors also.
* Initialization related errors can be found by using "graphicresult()" function.
* "graphicresult()" function returns error code, if the graphics are not initialized properly, if it is initialized properly then we will get "grOK".
* By using "grapherrormsg()" function we can display error message.
* By using "closegraph()" function we can close all properties of the graphics.

/\* C graphics program to draw a line \*/

#include<graphics.h>

#include<conio.h>

void main()

{

int gd = DETECT, gm;

/\* initialization of graphic mode \*/

initgraph(&gd, &gm, "");

line(100,100,200,200);

getch();

closegraph();

}

#include<graphics.h>

#include<conio.h>

void main()

{

int gd = DETECT, gm;

initgraph(&gd, &gm, "");

circle(300, 300, 50);

getch();

closegraph();

}

#include<graphics.h>

#include<stdio.h>

void main()

{

int gdriver = DETECT, gmode;

int x = 200, y = 200;

initgraph(&gdriver, &gmode, "");

outtextxy(x, y, "Hello World");

closegraph();

}

#include<graphics.h>

#include<conio.h>

void main()

{

int gd = DETECT, gm;

initgraph(&gd, &gm, "");

ellipse(100, 100, 0, 360, 50, 25);

getch();

closegraph();

}