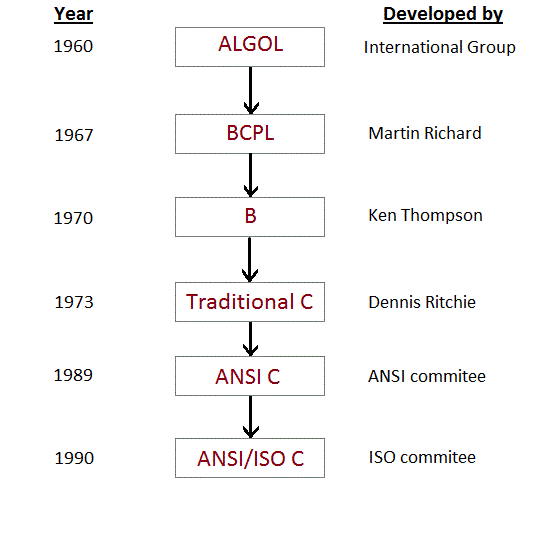
**Overview of C language**

C is a structured programming language developed by Dennis Ritchie in 1973 at Bell Laboratories. It is one of the most popular computer languages today because of its structure, high-level abstraction, machine independent feature. C language was developed with UNIX operating system, so it is strongly associated with UNIX, which is one of the most popular network operating system in use today and heart of internet data superhighway.

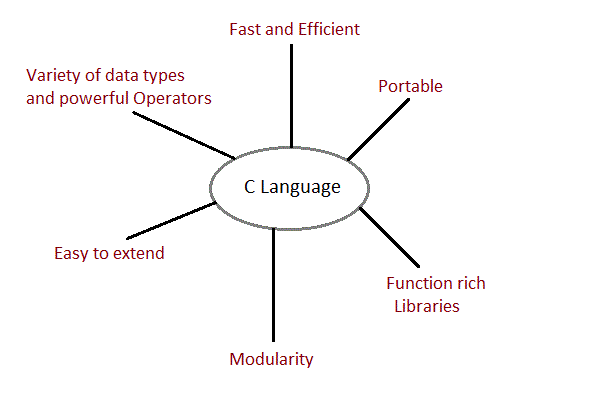
**History of C language**

C language has evolved from three different structured language ALGOL, BCPL and B Language. It uses many concepts from these languages and introduced many new concepts such as data types, struct, pointer. In 1988, the language was formalised by **American National Standard Institute**(ANSI). In 1990, a version of C language was approved by the **International Standard Organisation**(ISO) and that version of C is also referred to as C89.



**Features of C language**

* It is a robust language with rich set of built-in functions and operators that can be used to write any complex program.
* The C compiler combines the capabilities of an assembly language with features of a high-level language.
* Programs Written in C are efficient and fast. This is due to its variety of data type and powerful operators.
* It is many time faster than BASIC.
* C is highly portable this means that programs once written can be run on another machines with little or no modification.
* Another important feature of C program, is its ability to extend itself.
* A C program is basically a collection of functions that are supported by C library. We can also create our own function and add it to C library.
* C language is the most widely used language in operating systems and embedded system development today.



**First C language Program**

Lets see how to write a simple c program

#include <stdio.h>

#include <conio.h>

int main()

{

printf("Hello,World");

getch();

return 0;

}

**Different parts of C program.**

* Pre-processor
* Header file
* Function
* Variables
* expression
* Comment

All these are essential parts of a C language program.

**Pre-processor**

**#include**, the first word of any C program. It is also known as pre-processor. The main work of pre-processor is to initialize the environment of program, i.e to link the program with the header file**<stdio.h>< b="">.</stdio.h><>**

**Header file**

**Header file is a collection of built-in functions that help us in our program. Header files contain definitions of functions and variables which can be incorporated into any C program by pre-processor #include statement. Standard header files are provided with each compiler, and cover a range of areas like string handling, mathematical functions, data conversion, printing and reading of variables.**

**To use any of the standard functions, the appropriate header file must be included. This is done at the beginning of the C source file.**

**For example, to use the printf() function in a program, the line #include <stdio.h> is responsible.**

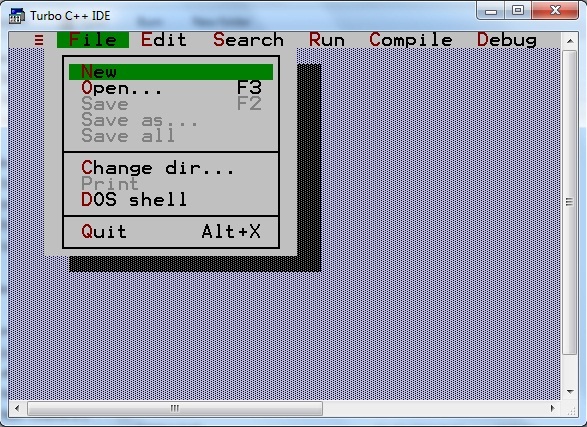
**main() function**

**main() function is a function that must be used in every C program. A function is a sequence of statement required to perform a specific task. main() function starts the execution of C program. In the above example, int in front of main() function is the return type of main() function. we will discuss about it in detail later. The curly braces { } just after the main() function encloses the body ofmain() function.**

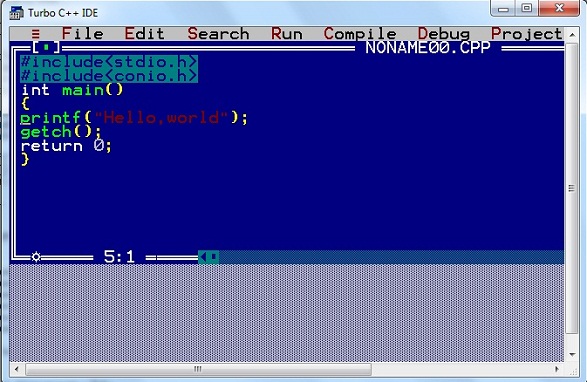
**Compile and Run**

**There are many different ways to compile and run a C program. All that is required is a C compiler. We will recommend you to use turbo c IDE, oldest IDE for c programming. It is freely available over internet and is good for a beginner.**

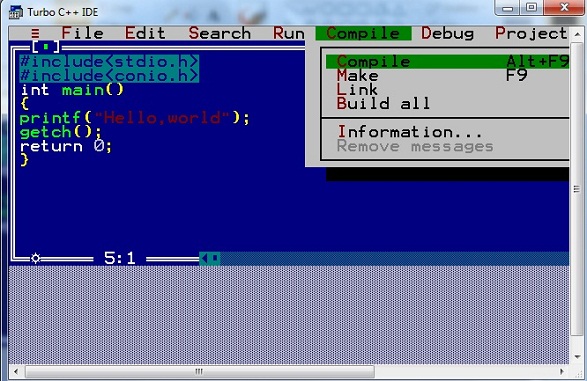
**Step 1 : Open turbo C IDE(Integrated Development Environment), click on File and then click on New**

****

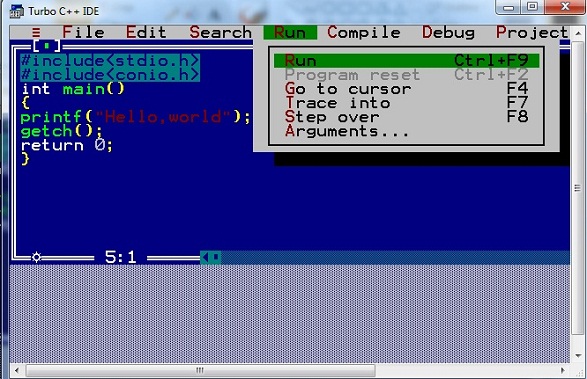
**Step 2 : Write the above example as it is**

****

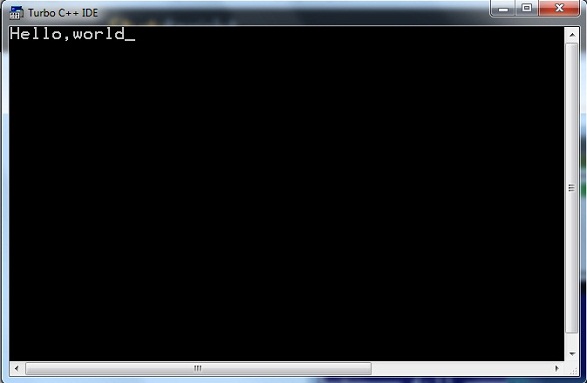
**Step 3 : Click on compile or press Alt+f9 to compile the code**

****

**Step 4 : Click on Run or press Ctrl+f9 to run the code**

****

**Step 5 : Output**

****

**C Input output function**

C programming language provides many of the built-in functions to read given input and write data on screen, printer or in any file.

**scanf() and printf() functions**

#include<stdio.h>

#include<conio.h>

void main()

{

int i;

printf("Enter a value");

scanf("%d",&i);

printf( "\nYou entered: %d",i);

getch();

}

When you will compile the above code, it will ask you to enter a value. When you will enter the value, it will display the value you have entered.

**NOTE :** printf() function returns the number of characters printed by it, and scanf() returns the number of characters read by it.

int i = printf("studytonight");

In this program i will get 12 as value, because studytonight has 12 characters.

**getchar() & putchar() functions**

The getchar() function reads a character from the terminal and returns it as an integer. This function reads only single character at a time. You can use this method in the loop in case you want to read more than one characters. The putchar() function prints the character passed to it on the screen and returns the same character. This function puts only single character at a time. In case you want to display more than one characters, use putchar() method in the loop.

#include <stdio.h>

#include <conio.h>

void main( )

{

int c;

printf("Enter a character");

c=getchar();

putchar(c);

getch();

}

When you will compile the above code,it will ask you to enter a value. When you will enter the value, it will display the value you have entered.

**gets() & puts() functions**

The gets() function reads a line from **stdin** into the buffer pointed to by **s** until either a terminating newline or EOF (end of file). The puts() function writes the string **s** and a trailing newline to stdout.

#include<stdio.h>

#include<conio.h>

void main()

{

char str[100];

printf("Enter a string");

gets( str );

puts( str );

getch();

}

When you will compile the above code,it will ask you to enter a string. When you will enter the string, it will display the value you have entered.

**Difference between scanf() and gets()**

The main difference between these two functions is that scanf() stops reading characters when it encounters a space, but gets() reads space as character too.

If you enter name as **Study Tonight** using scanf() it will only read and store **Study** and will leave the part after space. But gets() function will read it complete.

**C Language Syntax Rules**

C language syntax specify rules for sequence of characters to be written in C language. The rule specify how character sequence will be grouped together to form **tokens**. A smallest individual unit in c program is known as C Tokens. Tokens are either keyword, identifier, constant, variable or any symbol which has some meaning in C language. A C program can also be called as collection of various tokens.

Example of C tokens,

|  |  |  |  |
| --- | --- | --- | --- |
| int | curly braces { } | comments | semicolon ; |

**Comments**

Comments are simple text in your C program that increases readability of programs. Compiler ignore anything written as comment in your program.

**Example of comments :**

//This is a comment *Single line Comment*

/\*This is a comment\*/ *Single line Comment*

/\*This is a long

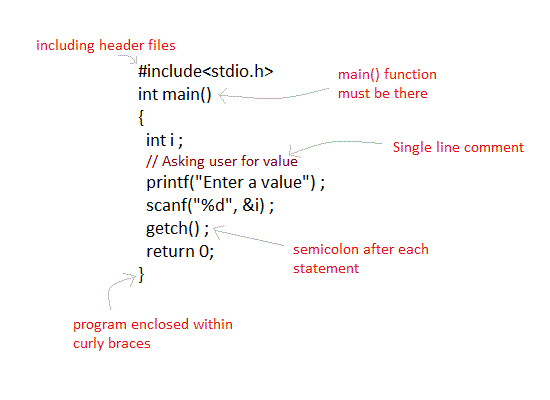
and valid comment\*/ *Multi line Comment*

//this is not

a valid comment Wrong Syntax

**Some basic syntax rule for C program**

* C is a case sensitive language so all C instructions must be written in lower case letter.
* All C statement must be end with a semicolon.
* Whitespace is used in C to describe blanks and tabs.
* Whitespace is required between keywords and identifiers



**Keywords**

Keywords are preserved words that 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 |
| const | extern | return | union |
| char | float | short | unsigned |
| continue | for | signed | volatile |
| default | goto | sizeof | void |
| do | if | static | while |

**Identifiers**

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**

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

**Character set**

In C language characters are grouped into the following catagories,

1. Letters(all alphabets a to z & A to Z).
2. Digits (all digits 0 to 9).
3. Special characters, ( such as colon :, semicolon ;, period ., underscore \_, ampersand & etc).
4. White spaces.

**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
* Bitwise operators
* Assignment operators
* Conditional operators
* Special operators

**Arithmetic operators**

C supports all the basic arithmetic operators. The following table shows all the basic arithmetic operators.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| + | adds two operands |
| - | subtract second operands from first |
| \* | multiply two operand |
| / | divide numerator by denumerator |
| % | remainder of division |
| ++ | Increment operator increases integer value by one |
| -- | Decrement operator decreases integer value by one |

**Relation operators**

The following table shows all relation operators supported by C.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| == | Check if two operand are equal |
| != | Check if two operand are not equal. |
| > | Check if operand on the left is greater than operand on the right |
| < | Check operand on the left is smaller than right operand |
| >= | check left operand is greater than or equal to right operand |
| <= | Check if operand on left is smaller than or equal to right operand |

**Logical operators**

C language supports following 3 logical operators. Suppose a=1 and b=0,

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| && | Logical AND | (a && b) is false |
| || | Logical OR | (a || b) is true |
| ! | Logical NOT | (!a) is false |

**Bitwise operators**

Bitwise operators perform manipulations of data at **bit level**. These operators also perform **shifting of bits**from right to left. Bitwise operators are not applied to **float** or **double**.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| & | Bitwise AND |
| | | Bitwise OR |
| ^ | Bitwise exclusive OR |
| << | left shift |
| >> | right shift |

Now lets see truth table for bitwise &, | and ^

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **a** | **b** | **a & b** | **a | b** | **a ^ b** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

The bitwise shift operators shifts the bit value. The left operand specifies the value to be shifted and the right operand specifies the number of positions that the bits in the value are to be shifted. Both operands have the same precedence.

*Example* :

a = 0001000

b= 2

a << b = 0100000

a >> b = 0000010

**Assignment Operators**

Assignment operators supported by C language are as follows.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | assigns values from right side operands to left side operand | a=b |
| += | adds right operand to the left operand and assign the result to left | a+=b is same as a=a+b |
| -= | subtracts right operand from the left operand and assign the result to left operand | a-=b is same as a=a-b |
| \*= | mutiply left operand with the right operand and assign the result to left operand | a\*=b is same as a=a\*b |
| /= | divides left operand with the right operand and assign the result to left operand | a/=b is same as a=a/b |
| %= | calculate modulus using two operands and assign the result to left operand | a%=b is same as a=a%b |

**Conditional operator**

It is also known as ternary operator and used to evaluate conditional expression.

epr1 ? expr2 : expr3

If **epr1** Condition is true ? Then value **expr2** : Otherwise value **expr3**

**Special operator**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| sizeof | Returns the size of an variable | **sizeof(x)** return size of the variable **x** |
| & | Returns the address of an variable | **&x ;** return address of the variable **x** |
| \* | Pointer to a variable | **\*x ;** will be pointer to a variable **x** |

**Data types in C Language**

Data types specify how we enter data into our programs and what type of data we enter. C language has some predefined set of data types to handle various kinds of data that we use in our program. These datatypes have different storage capacities.

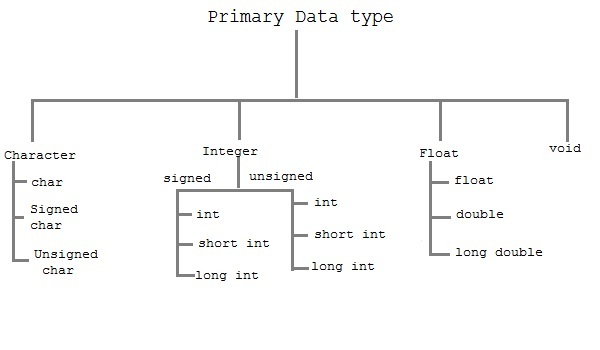
C language supports 2 different type of data types,

**Primary data types**

These are fundamental data types in C namely integer(**int**), floating(**float**), charater(**char**) and **void**.

**Derived data types**

Derived data types are like arrays, functions, stuctures and pointers. These are dicussed in detail later.



**Integer type**

Integers are used to store whole numbers.

**Size and range of Integer type on 16-bit machine**

|  |  |  |
| --- | --- | --- |
| **Type** | **Size(bytes)** | **Range** |
| int or signed int | 2 | -32,768 to 32767 |
| unsigned int | 2 | 0 to 65535 |
| short int or signed short int | 1 | -128 to 127 |
| long int or signed long int | 4 | -2,147,483,648 to 2,147,483,647 |
| unsigned long int | 4 | 0 to 4,294,967,295 |

**Floating type**

Floating types are used to store real numbers.

**Size and range of Integer type on 16-bit machine**

|  |  |  |
| --- | --- | --- |
| **Type** | **Size(bytes)** | **Range** |
| Float | 4 | 3.4E-38 to 3.4E+38 |
| double | 8 | 1.7E-308 to 1.7E+308 |
| long double | 10 | 3.4E-4932 to 1.1E+4932 |

**Character type**

Character types are used to store characters value.

**Size and range of Integer type on 16-bit machine**

|  |  |  |
| --- | --- | --- |
| **Type** | **Size(bytes)** | **Range** |
| char or signed char | 1 | -128 to 127 |
| unsigned char | 1 | 0 to 255 |

**void type**

void type means no value. This is usually used to specify the type of functions.

**Variables in C Language**

A variable is a name that may be used to store a data value. Unlike constant, variables are changeable, we can change value of a variable during execution of a program. A programmer can choose a meaningful variable name. Example : average, height, age, total etc.

**Rules to define variable name**

1. Variable name must be upto 8 characters.
2. Variable name must not start with a digit.
3. Variable name can consist of alphabets, digits and special symbols like underscore \_.
4. Blank or spaces are not allowed in variable name.
5. Keywords are not allowed as variable name.

**Declaration of variable**

Declaration of variables must be done before they are used in the program. Declaration does two things.

1. It tells the compiler what the variable name is.
2. It specifies what type of data the variable will hold.

#include<stdio.h>

#include<conio.h>

void main()

{

int a,b,sum; *//variable declaraction*

a=10;

b=20;

sum=a+b;

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

getch();

}

**Decision making in C**

Decision making is about deciding the order of execution of statements based on certain conditions or repeat a group of statements until certain specified conditions are met. C language handles decision-making by supporting the following statements,

* *if* statement
* *switch* statement
* conditional operator statement
* *goto* statement

**Decision making with *if* statement**

The *if* statement may be implemented in different forms depending on the complexity of conditions to be tested. The different forms are,

1. Simple *if* statement
2. *If....else* statement
3. Nested *if....else*statement
4. *else if* statement

**Simple *if* statement**

The general form of a simple *if* statement is,

if( expression )

{

statement-inside;

}

statement-outside;

If the *expression* is true, then 'statement-inside' it will be executed, otherwise 'statement-inside' is skipped and only 'statement-outside' is executed.

**Example :**

#include <stdio.h>

void main( )

{

int x,y;

x=15;

y=13;

if (x > y )

{

printf("x is greater than y");

}

}

**output:** x is greater than y

***if...else* statement**

The general form of a simple *if...else* statement is,

if( expression )

{

statement-block1;

}

else

{

statement-block2;

}

If the 'expression' is true, the 'statement-block1' is executed, else 'statement-block1' is skipped and 'statement-block2' is executed.

**Example :**

#include <stdio.h>

void main( )

{

int x,y;

x=15;

y=18;

if (x > y )

{

printf("x is greater than y");

}

else

{

printf("y is greater than x");

}

}

**Output:**

y is greater than x

**Nested *if....else* statement**

The general form of a nested *if...else* statement is,

if( expression )

{

if( expression1 )

{

statement-block1;

}

else

{

statement-block 2;

}

}

else

{

statement-block 3;

}

if 'expression' is false the 'statement-block3' will be executed, otherwise it continues to perform the test for 'expression 1' . If the 'expression 1' is true the 'statement-block1' is executed otherwise 'statement-block2' is executed.

**Example :**

#include <stdio.h>

#include <conio.h>

void main( )

{

int a,b,c;

clrscr();

printf("enter 3 number");

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

if(a>b)

{

if( a > c)

{

printf("a is greatest");

}

else

{

printf("c is greatest");

}

}

else

{

if( b> c)

{

printf("b is greatest");

}

else

{

printf("c is greatest");

}

}

getch();

}

***else-if* ladder**

The general form of else-if ladder is,

if(expression 1)

{

statement-block1;

}

else if(expression 2)

{

statement-block2;

}

else if(expression 3 )

{

statement-block3;

}

else

default-statement;

The expression is tested from the top(of the ladder) downwards. As soon as the true condition is found, the statement associated with it is executed.

**Example :**

#include <stdio.h>

#include <conio.h>

void main( )

{

int a;

printf("enter a number");

scanf("%d",&a);

if( a%5==0 && a%8==0)

{

printf("divisible by both 5 and 8");

}

else if( a%8==0 )

{

printf("divisible by 8");

}

else if(a%5==0)

{

printf("divisible by 5");

}

else

{

printf("divisible by none");

}

getch();

}

**Points to Remember**

1. In *if* statement, a single statement can be included without enclosing it into curly braces { }
2. int a = 5;
3. if(a > 4)
4. printf("success");

No curly braces are required in the above case, but if we have more than one statement inside *if* condition, then we must enclose them inside curly braces.

1. == must be used for comparison in the expression of *if* condition, if you use = the expression will always return true, because it performs assignment not comparison.
2. Other than **0(zero)**, all other values are considered as true.
3. if(27)

printf("hello");

**Switch statement**

Switch statement is used to solve multiple option type problems for menu like program, where one value is associated with each option. The **expression** in switch case evaluates to return an integral value, which is then compared to the values in different cases, where it matches that block of code is executed, if there is no match, then default block is executed. The general form of **switch** statement is,

switch(**expression**)

{

case value-1:

block-1;

break;

case value-2:

block-2;

break;

case value-3:

block-3;

break;

case value-4:

block-4;

break;

default:

default-block;

break;

}

**Points to Remember**

1. We don't use those expressions to evaluate switch case, which may return floating point values or strings.
2. It isn't necessary to use **break** after each block, but if you do not use it, all the consecutive block of codes will get executed after the matching block.
3. int i = 1;
4. switch(i)
5. {
6. case 1:
7. printf("A"); // No break
8. case 2:
9. printf("B"); // No break
10. case 3:
11. printf("C");
12. break;
13. }

Output : A B C

The output was supposed to be only **A** because only the first case matches, but as there is no break statement after the block, the next blocks are executed, until the cursor encounters a break.

1. **default** case can be placed anywhere in the switch case. Even if we don't include the default case switch statement works.

**Example of Switch Statement**

#include<stdio.h>

#include<conio.h>

void main( )

{

int a,b,c,choice;

clrscr( );

while(choice!=3)

{

printf("\n 1. Press 1 for addition");

printf("\n 2. Press 2 for subtraction");

printf("\n Enter your choice");

scanf("%d",&choice);

switch(choice)

{

**case 1:**

printf("Enter 2 numbers");

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

c=a+b;

printf("%d",c);

break;

**case 2:**

printf("Enter 2 numbers");

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

c=a-b;

printf("%d",c);

break;

**default:**

printf("you have passed a wrong key");

printf("\n press any key to continue");

}

}

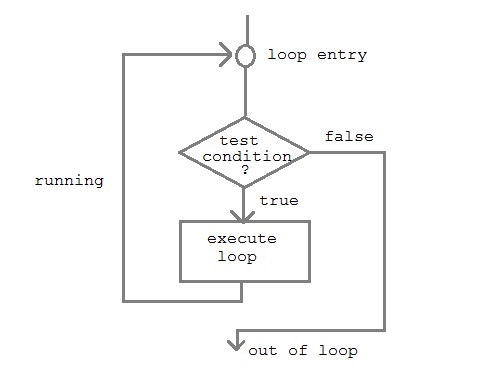
getch();

}

**How to use Loops in C Lanugage**

In any programming language, loops are used to execute a set of statements repeatedly until a particular condition is satisfied.

**How it Works**



A sequence of statements are executed until a specified condition is true. This sequence of statements to be executed is kept inside the curly braces { } known as the **Loop body**. After every execution of loop body, condition is verified, and if it is found to be **true** the loop body is executed again. When the condition check returns **false**, the loop body is not executed.

**There are 3 type of Loops in C language**

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

**while loop**

**while** loop can be addressed as an **entry control** loop. It is completed in 3 steps.

* Variable initialization.( e.g int x=0; )
* condition( e.g while( x<=10) )
* Variable increment or decrement ( x++ or x-- or x=x+2 )

**Syntax :**

variable initialization ;

while (condition)

{

statements ;

variable increment or decrement ;

}

**Example : Program to print first 10 natural numbers**

#include<stdio.h>

#include<conio.h>

void main( )

{

int x;

x=1;

**while**(x<=10)

{

printf("%d\t", x);

x++;

}

getch();

}

**output**

1 2 3 4 5 6 7 8 9 10

**for loop**

**for** loop is used to execute a set of statements repeatedly until a particular condition is satisfied. we can say it an **open ended loop.** General format is,

for(**initialization**; **condition** ; **increment/decrement**)

{

statement-block;

}

In **for** loop we have exactly two semicolons, one after initialization and second after condition. In this loop we can have more than one initialization or increment/decrement, separated using comma operator. **for** loop can have only one **condition**.

**Above Example with for loop**

#include<stdio.h>

#include<conio.h>

void main( )

{

int x;

**for**(x=1; x<=10; x++)

{

printf("%d\t",x);

}

getch();

}

**Output**

1 2 3 4 5 6 7 8 9 10

**Nested for loop**

We can also have nested **for** loops, i.e one **for** loop inside another **for** loop. Basic syntax is,

for(**initialization**; **condition**; **increment/decrement**)

{

for(**initialization**; **condition**; **increment/decrement**)

{

statement ;

}

}

**Example : Program to print half Pyramid of numbers**

#include<stdio.h>

#include<conio.h>

void main( )

{

int i,j;

**for**(i=1;i<5;i++)

{

printf("\n");

**for**(j=i;j>0;j--)

{

printf("%d",j);

}

}

getch();

}

**output**

1

21

321

4321

54321

**do while loop**

In some situations it is necessary to execute body of the loop before testing the condition. Such situations can be handled with the help of **do-while** loop. **do** statement evaluates the body of the loop first and at the end, the condition is checked using **while** statement. General format of **do-while** loop is,

do

{

....

.....

}

while(condition)

**Example : Program to print first ten multiple of 5.**

#include<stdio.h>

#include<conio.h>

void main()

{

int a,i;

a=5;

i=1;

**do**

{

printf("%d\t",a\*i);

i++;

}

**while**(i <= 10);

getch();

}

**output**

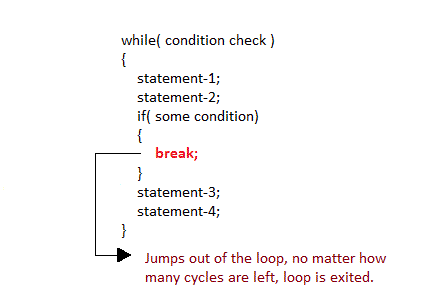
5 10 15 20 25 30 35 40 45 50

**Jumping Out of Loops**

Sometimes, while executing a loop, it becomes necessary to skip a part of the loop or to leave the loop as soon as certain condition becomes true, that is called jumping out of loop. C language allows jumping from one statement to another within a loop as well as jumping out of the loop.

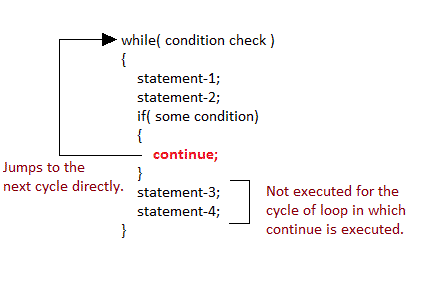
**1) break statement**

When **break** statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop.



**2) continue statement**

It causes the control to go directly to the test-condition and then continue the loop process. On encountering continue, cursor leave the current cycle of loop, and starts with the next cycle.



**Arrays**

In C language, arrays are reffered to as structured data types. An array is defined as **finite ordered collection of homogenous** data, stored in contiguous memory locations.

Here the words,

* **finite** *means* data range must be defined.
* **ordered** *means* data must be stored in continuous memory addresses.
* **homogenous** *means* data must be of similar data type.

**Example where arrays are used,**

* to store list of Employee or Student names,
* to store marks of a students,
* or to store list of numbers or characters etc.

Since arrays provide an easy way to represent data, it is classified amongst the data structures in C. Other data structures in c are **structure**, **lists**, **queues** and **trees**. Array can be used to represent not only simple list of data but also table of data in two or three dimensions.

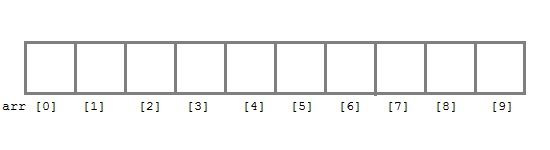
**Declaring an Array**

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

data-type variable-name[size];

**for example :**

int arr[10];



Here **int** is the data type, **arr** is the name of the array and 10 is the size of array. It means array **arr** can only contain 10 elements of **int** type. **Index** of an array starts from 0 to size-1 i.e first element of **arr** array will be stored at arr[0] address and last element will occupy arr[9].

**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**

Compile time initializtion of array elements is same as ordinary variable initialization. The general form of initialization of array is,

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

int marks[4]={ 67, 87, 56, 77 }; *//integer array initialization*

float area[5]={ 23.4, 6.8, 5.5 }; *//float array initialization*

int marks[4]={ 67, 87, 56, 77, 59 }; //Compile time error

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

#include<stdio.h>

#include<conio.h>

void main()

{

int i;

int arr[]={2,3,4}; *//Compile time array initialization*

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

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

}

getch();

}

**Output :**

2 3 4

**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. Example,

#include<stdio.h>

#include<conio.h>

void main()

{

int arr[4];

int i, j;

printf("Enter array element");

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

{

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

}

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

{

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

}

getch();

}

**Two dimensional Arrays**

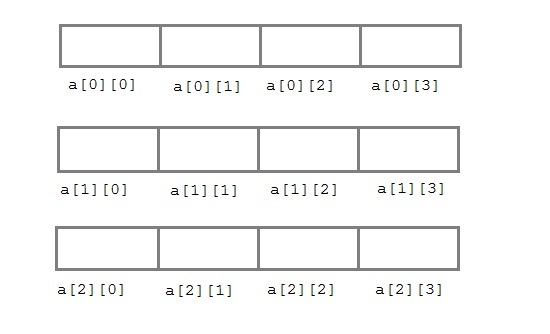
C language supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.

Two-dimensional array is declared as follows,

type array-name[row-size][column-size]

**Example :**

int a[3][4];



The above array can also be declared and initialized together. Such as,

int arr[][3] = {

{0,0,0},

{1,1,1}

};

**Run-time initialization of two dimensional Array**

#include<stdio.h>

#include<conio.h>

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]);

}

}

getch();

}

**string and Character array**

**string** is a sequence of characters that is treated as a single data item and terminated by null character'\0'. Remember that C language does not support strings as a data type. A **string** is actually one-dimensional array of characters in C language. These are often used to create meaningful and readable programs.

**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**

There are different ways to initialize a character array variable.

char name[10]="StudyTonight"; *//valid character array initialization*

char name[10]={'L','e','s','s','o','n','s','\0'}; *//valid initialization*

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

Some examples of illegal initialization of character array are,

char ch[3]="hell"; //Illegal

char str[4];

str="hell"; //Illegal

**String Input and Output**

Input function scanf() can be used with **%s** format specifier to read a string input from the terminal. But there is one problem with **scanf()** function, it terminates its input on first white space it encounters. Therefore if you try to read an input string "Hello World" using **scanf()** function, it will only read **Hello** and terminate after encountering white spaces.

However, C supports a format specification known as the **edit set conversion code %[..]** that can be used to read a line containing a variety of characters, including white spaces.

#include<stdio.h>

#include<conio.h>

#include<string.h>

void main()

{

char str[20];

clrscr();

printf("Enter a string");

scanf("%[^\n]",&str);

printf("%s",str);

getch();

}

Another method to read character string with white spaces from terminal is **gets()** function.

char text[20];

gets(text);

printf("%s",text);

**String Handling Functions**

C language supports a large number of string handling functions that can be used to carry out many of the string manipulations. These functions are packaged in **string.h** library. Hence, you must include **string.h**header file in your program to use these functions.

The following are the most commonly used string handling functions.

|  |  |
| --- | --- |
| **Method** | **Description** |
| strcat() | It is used to concatenate(combine) two string |
| 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 |

**strcat() function**

strcat("hello","world");

strcat() function will add the string **"world"** to **"hello"**.

**strlen() function**

strlen() function will return the length of the string passed to it.

int j;

j=strlen("studytonight");

printf("%d",j);

**output :**

12

**strcmp() function**

strcmp() function will return the ASCII difference between first unmatching character of two strings.

int j;

j=strcmp("study","tonight");

printf("%d",j);

**output:**

-1

**Storage classes**

In C language, each variable has a **storage class** which decides scope, visibility and lifetime of that variable. The following storage classes are most oftenly used in C programming,

1. **Automatic variables**
2. **External variables**
3. **Static variables**
4. **Register variables**

**Automatic variables**

A variable declared inside a function without any storage class specification, is by default an **automatic variable**. They are created when a function is called and are destroyed **automatically** when the function exits. Automatic variables can also be called local variables because they are local to a function. By default they are assigned **garbage value** by the compiler.

void main()

{

int detail;

or

auto int detail; *//Both are same*

}

**External or Global variable**

A variable that is declared outside any function is a **Global variable**. **Global** variables remain available throughout the entire program. One important thing to remember about global variable is that their values can be changed by any function in the program.

int number;

void main()

{

number=10;

}

fun1()

{

number=20;

}

fun2()

{

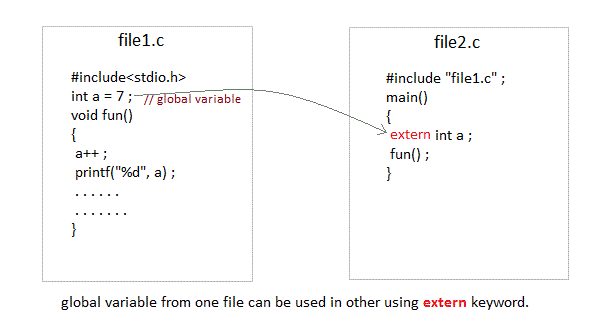
number=30;

}

Here the global variable **number** is available to all three functions.

**extern keyword**

The **extern** keyword is used before a variable to inform the compiler that this variable is declared somewhere else. The **extern** declaration does not allocate storage for variables.



**Problem when extern is not used**

main()

{

a = 10; //Error:cannot find variable a

printf("%d",a);

}

**Example Using extern in same file**

main()

{

**extern** int x; *//Tells compiler that it is defined somewhere else*

x = 10;

printf("%d",x);

}

int x;  *//Global variable x*

**Static variables**

A **static** variable tells the compiler to persist the variable until the end of program. Instead of creating and destroying a variable every time when it comes into and goes out of scope, **static** is initialized only once and remains into existence till the end of program. A static variable can either be internal or external depending upon the place of declaraction. Scope of **internal static** variable remains inside the function in which it is defined. **External static** variables remain restricted to scope of file in each they are declared.

They are assigned **0 (zero)** as default value by the compiler.

void test(); //Function declaration (discussed in next topic)

main()

{

test();

test();

test();

}

void test()

{

static int a = 0; //Static variable

a = a+1;

printf("%d\t",a);

}

**output :**

1 2 3

**Register variable**

**Register** variable inform the compiler to store the variable in register instead of memory. **Register** variable has faster access than normal variable. Frequently used variables are kept in register. Only few variables can be placed inside register.

**NOTE :**We can never get the address of such variables.

**Syntax :**

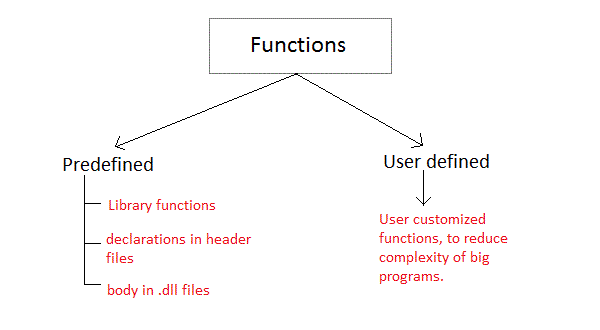
register int number;

**Functions in c**

A function is a block of code that performs a particular task. There are times when we need to write a particular block of code for more than once in our program. This may lead to bugs and irritation for the programmer. C language provides an approach in which you need to declare and define a group of statements once and that can be called and used whenever required. This saves both time and space.

C functions can be classified into two categories,

* **Library functions**
* **User-defined 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** 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**

1. It provides modularity to the program.
2. Easy code Reuseability. You just have to call the function by its name to use it.
3. In case of large programs with thousands of code lines, debugging and editing becomes easier if you use functions.

**Function declaration**

General syntax of function declaration is,

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

Like variable and an array, a function must also be declared before its called. A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately. A function declaration consist of 4 parts.

* return-type
* function name
* parameter list
* terminating semicolon

**Function definition Syntax**

General syntax of function definition is,

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

{

function-body ;

}

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

**return-type**

return type specifies the type of value(int,float,char,double) that function is expected to return to the program calling the function.

**function-name**

function name specifies the name of the function. The function name is any valid C identifier and therefore must follow the same rule of formation as other variables in C.

**parameter-list**

The parameter list declares the variables that will receive the data sent by calling program. They often referred to as formal parameters. These parameters are also used to send values to calling program.

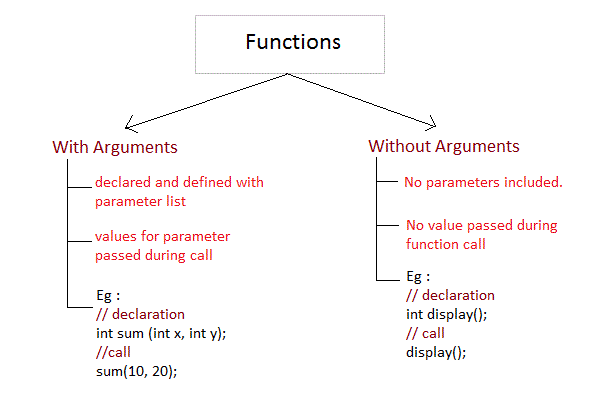
**function-body**

The function body contains the declarations and the statement(algorithm) necessary for performing the required task. The body is enclosed within curly braces { } and consists of three parts.

* **local** variable declaration.
* **function statement** that performs the tasks of the function.
* a **return** statement that return the value evaluated by the function.

**Functions and Arguments**

Arguments are the values specified during the function call, for which the formal parameters are declared in the function.



**Example : Function that return some value**

#include<stdio.h>

#include<conio.h>

int larger(int a,int b); // function declaration

void main()

{

int i,j,k;

clrscr();

i=99;

j=112;

k=larger(i,j); // function call

printf("%d",k);

getch();

}

int larger(int a,int b) // function declaration

{

if(a>b)

return a;

else

return b;

}

**Nesting of Functions**

C language also allows nesting of functions, one function using another function inside its body. We must be careful while using nested functions, because it may lead to infinte nesting.

**function1()**

{

**function2() ;**

//statements

}

If function2 calls function1 inside it, then in this case it will lead to infinite nesting, they will keep calling each other. Hence we must be careful.

**Recursion**

Recursion is a special of nesting functions, where a function calls itself inside it. We must have certain condition to break out of the recursion, otherwise recursion is infinite.

**function1()**

{

**function1() ;**

//statements

}

**Example : Factorial of a number using Recursion**

#include<stdio.h>

#include<conio.h>

int factorial(int x);

void main()

{

int a,b;

clrscr();

printf("Enter no.");

scanf("%d",&a);

b=factorial(a);

printf("%d",b);

getch();

}

int factorial(int x)

{

int r=1;

if(x==1) return 1;

else r=x\*factorial(x-1);

return r;

}

**Types of Function calls in C**

Functions are called by their names. If the function is without argument, it can be called directly using its name. But for functions with arguments, we have two ways to call them,

1. Call by Value
2. Call by Reference

**Call by Value**

In this calling technique we pass the values of arguments which are stored or copied into the formal parameters of functions. Hence, the original values are unchanged only the parameters inside function changes.

void **calc**(int x);

int main()

{

int x = 10;

**calc**(x);

printf("%d", x);

}

void **calc**(int x)

{

x = x + 10 ;

}

Output : 10

In this case the actual variable x is not changed, because we pass argument by value, hence a copy of x is passed, which is changed, and that copied value is destroyed as the function ends(goes out of scope). So the variable **x** inside main() still has a value 10.

But we can change this program to modify the original **x**, by making the function **calc()** return a value, and storing that value in x.

int **calc**(int x);

int main()

{

int x = 10;

x = **calc**(x);

printf("%d", x);

}

int **calc**(int x)

{

x = x + 10 ;

return x;

}

Output : 20

**Call by Reference**

In this we pass the address of the variable as arguments. In this case the formal parameter can be taken as a reference or a pointer, in both the case they will change the values of the original variable.

void **calc**(int \*p);

int main()

{

int x = 10;

**calc**(&x); // passing address of x as argument

printf("%d", x);

}

void **calc**(int \*p)

{

\*p = \*p + 10;

}

Output : 20

**Passing Array to Function**

Whenever we need to pass a list of elements as argument to the function, it is prefered to do so using an array. But how can we pass an array as argument ? Lets see how to do it.

**Declaring Function with array in parameter list**

There are two possible ways to do so, one will lead to call by value and the other is used to perform call be reference.

* We can either have an array in parameter.
* int sum (int arr[]);
* Or, we can have a pointer in the parameter list, to hold base address of array.
* int sum (int\* ptr);

We will study this in detail later when we will study pointers.

**Returning Array from function**

We dont't return an array from functions, rather we return a pointer holding the base address of the array to be returned. But we must, make sure that the array exists after the function ends.

int\* sum (int x[])

{

//statements

return x ;

}

We will discuss about this when we will study pointers with arrays.

**Example : Create a function to sort an Array of elements**

void sorting(int x[],int y)

{

int i, j, temp ;

for(i=1; i<=y-1; i++)

{

for(j=0; j< y-i; j++)

{

if(x[j] > x[j+1])

{

temp = x[j];

x[j] = x[j+1];

x[j+1] = temp;

}

}

}

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

{

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

}

}

In the above example,

* return type is **void**, that means function does not return any thing.
* **Sorting** is the function name.
* int x[ ] and int y is the **parameter list**.
* int i, j, temp inside curly braces are the local variable declaraction.

**Introduction to Structure**

* Structure is a user-defined data type in C which allows you to combine different data types to store a particular type of record. Structure helps to construct a complex data type in more meaningful way. It is somewhat similar to an Array. The only difference is that array is used to store collection of similar datatypes 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*

{

//Statements

};

**Example of Structure**

struct **Book**

{

char name[15];

int price;

int pages;

};

* Here the **struct Book** declares a structure to hold the details of book which consists of three data fields, namely *name*, *price* and *pages*. These fields are called **structure elements or members**. Each member can have different data type,like in this case, **name** is of char type and **price** is of int type etc. **Book** is the name of the structure and is called structure tag.
* **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.

**1) Declaring Structure variables separately**

struct Student

{

char[20] name;

int age;

int rollno;

} ;

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

**2) Declaring Structure Variables with Structure definition**

struct **Student**

{

char[20] name;

int age;

int rollno;

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

* Here **S1** and **S2** are variables of structure **Student**. However this approach is not much recommended.

**Accessing Structure Members**

* Structure members can be accessed and assigned values in number of ways. Structure member has no meaning independently. In order to assign a value to a structure member, the member name must be linked with the **structure** variable using dot . operator also called **period** or **member access** operator.

struct Book

{

char name[15];

int price;

int pages;

} b1 , b2 ;

**b1.price=200;** *//b1 is variable of Book type and price is member of Book*

* We can also use scanf() to give values to structure members through terminal.

scanf(" %s ", b1.name);

scanf(" %d ", &b1.price);

**Structure Initialization**

* Like any other data type, structure variable can also be initialized at compile time.

struct **Patient**

{

float height;

int weight;

int age;

};

struct Patient **p1** = { 180.75 , 73, 23 }; *//initialization*

or,

**struct patient p1;**

**p1.height = 180.75; *//initialization of each member separately***

**p1.weight = 73;**

**p1.age = 23;**

**Array of Structure**

* We can also declare an array of **structure**. Each element of the array representing a **structure** variable.**Example :** struct employee emp[5];
* The above code define an array **emp** of size 5 elements. Each element of array **emp** is of type **employee**

#include<stdio.h>

#include<conio.h>

struct employee

{

char ename[10];

int sal;

};

struct employee emp[5];

int i,j;

void ask()

{

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

{

printf("\nEnter %dst employee record\n",i+1);

printf("\nEmployee name\t");

scanf("%s",emp[i].ename);

printf("\nEnter employee salary\t");

scanf("%d",&emp[i].sal);

}

printf("\nDisplaying Employee record\n");

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

{

printf("\nEmployee name is %s",emp[i].ename);

printf("\nSlary is %d",emp[i].sal);

}

}

void main()

{

clrscr();

ask();

getch();

}

**Nested Structures**

* Nesting of structures, is also permitted in C language.

**Example :**

struct **student**

{

char[30] name;

int age;

struct **address**

{

char[50] locality;

char[50] city;

int pincode;

};

};

**Structure as function arguments**

* We can pass a structure as a function argument in similar way as we pass any other variable or array.

**Example :**

#include<stdio.h>

#include<conio.h>

struct student

{

char name[10];

int roll;

};

void show(struct student st);

void main()

{

struct student std;

clrscr();

printf("\nEnter student record\n");

printf("\nstudent name\t");

scanf("%s",std.name);

printf("\nEnter student roll\t");

scanf("%d",&std.roll);

show(std);

getch();

}

void show(struct student st)

{

printf("\nstudent name is %s",st.name);

printf("\nroll is %d",st.roll);

}

**typedef**

**typedef** is a keyword used in C language to assign alternative names to existing types. Its mostly used with user defined data types, when names of data types get slightly complicated. Following is the general syntax for using typedef,

**typedef** *existing\_name* *alias\_name*

Lets take an example and see how typedef actually works.

typedef unsigned long ulong;

The above statement define a term **ulong** for an unsigned long type. Now this **ulong** identifier can be used to define unsigned long type variables.

ulong i, j ;

**Application of typedef**

**typedef** can be used to give a name to user defined data type as well. Lets see its use with structures.

**typedef** struct

{

type member1;

type member2;

type member3;

} **type\_name** ;

Here **type\_name** represents the stucture definition associated with it. Now this **type\_name** can be used to declare a variable of this stucture type.

type\_name t1, t2 ;

**Example of structure definition using typedef**

#include<stdio.h>

#include<conio.h>

#include<string.h>

typedef struct employee

{

char name[50];

int salary;

} **emp** ;

void main( )

{

emp e1;

printf("\nEnter Employee record\n");

printf("\nEmployee name\t");

scanf("%s",e1.name);

printf("\nEnter Employee salary \t");

scanf("%d",&e1.salary);

printf("\nstudent name is %s",e1.name);

printf("\nroll is %d",e1.salary);

getch();

}

**typedef and Pointers**

typedef can be used to give an alias name to pointers also. Here we have a case in which use of typedef is beneficial during pointer declaration.

In Pointers \* binds to the right and not the left.

int\* x, y ;

By this declaration statement, we are actually declaring **x** as a pointer of type int, whereas **y** will be declared as a plain integer.

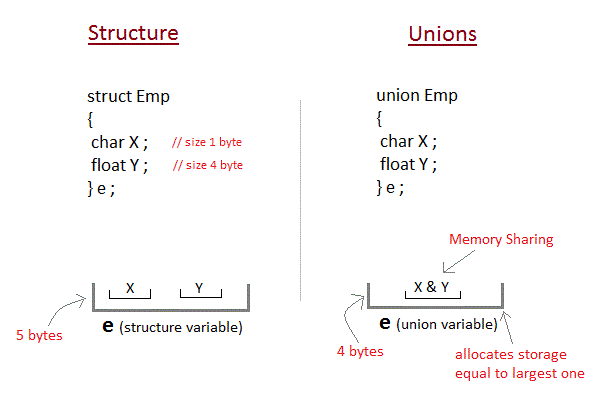
typedef int\* **IntPtr** ;

**IntPtr** x, y, z;

But if we use **typedef** like in above example, we can declare any number of pointers in a single statement.

**Unions in C Language**

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



This implies that although a **union** may contain many members of different types, **it cannot handle all the members at same time**. A **union** is declared using **union** keyword.

**union** item

{

int m;

float x;

char c;

}It1;

This declares a variable **It1** of type union**item**. This **union** contains three members each with a different data type. However only one of them can be used at a time. This is due to the fact that only one location is allocated for a **union** variable, irrespective of its size. The compiler allocates the storage that is large enough to hold largest variable type in the **union**. In the **union** declared above the member **x** requires 4 bytes which is largest among the members in 16-bit machine. Other members of **union** will share the same address.

**Accessing a Union Member**

Syntax for accessing **union** member is similar to accessing structure member,

**union** test

{

int a;

float b;

char c;

}t;

t.a ; *//to access members of union t*

t.b ;

t.c ;

**Complete Example for Union**

#include <stdio.h>

#include <conio.h>

union item

{

int a;

float b;

char ch;

};

int main( )

{

union item it;

it.a = 12;

it.b = 20.2;

it.ch='z';

clrscr();

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

printf("%f\n",it.b);

printf("%c\n",it.ch);

getch();

return 0;

}

**Output :**

-26426

20.1999

z

As you can see here, the values of **a** and **b** get corrupted and only variable **c** prints the expected result. Because in **union**, the only member whose value is currently stored will have the memory.

**Introduction to Pointers**

Pointers are variables that hold address of another variable of same data type.

Pointers are one of the most distinct and exciting features of C language. It provides power and flexibility to the language. Although pointer may appear little confusing and complicated in the beginning, but trust me its a powerful tool and handy to use once its mastered.

**Benefit of using pointers**

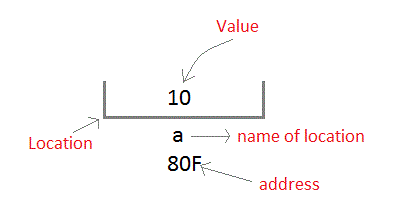
* Pointers are more efficient in handling Array and Structure.
* Pointer allows references to function and thereby helps in passing of function as arguments to other function.
* It reduces length and the program execution time.
* It allows C to support dynamic memory management.

**Concept of Pointer**

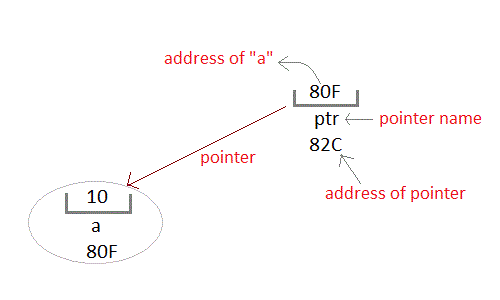
Whenever a **variable** is declared, system will allocate a location to that variable in the memory, to hold value. This location will have its own address number.

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

int a = 10 ;



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



**Declaring a pointer variable**

General syntax of pointer declaration is,

*data-type* *\*pointer\_name;*

Data type of pointer must be same as the variable, which the pointer is pointing. **void** type pointer works with all data types, but isn't used oftenly.

**Initialization of Pointer variable**

**Pointer Initialization** is the process of assigning address of a variable to **pointer** variable. Pointer variable contains address of variable of same data type. In C language **address operator** & is used to determine the address of a variable. The & (immediately preceding a variable name) returns the address of the variable associated with it.

int a = 10 ;

int \*ptr ; *//pointer declaration*

ptr = &a ; *//pointer initialization*

or,

int \*ptr = &a ; *//initialization and declaration together*

Pointer variable always points to same type of data.

float a;

int \*ptr;

ptr = &a; //ERROR, type mismatch

**Dereferencing of Pointer**

Once a pointer has been assigned the address of a variable. To access the value of variable, pointer is dereferenced, using the **indirection operator** \*.

int a,\*p;

a = 10;

p = &a;

printf("%d",\*p); *//this will print the value of a.*

printf("%d",\*&a); *//this will also print the value of a.*

printf("%u",&a); *//this will print the address of a.*

printf("%u",p); *//this will also print the address of a.*

printf("%u",&p); *//this will also print the address of p.*

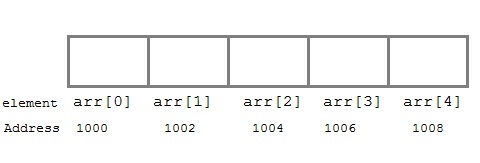
**Pointer and Arrays**

When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array. Base address which gives location of the first element is also allocated by the compiler.

Suppose we declare an array **arr**,

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

Assuming that the base address of **arr** is 1000 and each integer requires two byte, the five element will be stored as follows



Here variable **arr** will give the base address, which is a constant pointer pointing to the element, **arr[0]**. Therefore **arr** is containing the address of **arr[0]** i.e 1000.

**arr** *is equal to* **&arr[0]** // by default

We can declare a pointer of type int to point to the array **arr**.

int \*p;

p = arr;

or p = &arr[0]; //both the statements are equivalent.

Now we can access every element of array **arr** using **p++** to move from one element to another.

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

**Pointer to Array**

As studied above, we can use a pointer to point to an Array, and then we can use that pointer to access the array. Lets have an example,

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++)

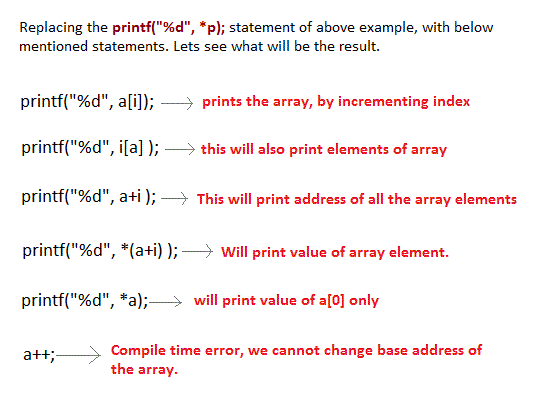
{

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

p++;

}

In the aboce program, the pointer **\*p** will print all the values stored in the array one by one. We can also use the Base address (**a** in above case) to act as pointer and print all the values.



**Pointer to Multidimensional Array**

A multidimensional array is of form, a[i][j]. Lets see how we can make a pointer point to such an array. As we know now, name of the array gives its base address. In a[i][j], **a** will give the base address of this array, even a+0+0 will also give the base address, that is the address of **a[0][0]** element.

Here is the generalized form for using pointer with multidimensional arrays.

**\*(\*(ptr + i) + j)** *is same as* **a[i][j]**

**Pointer and Character strings**

Pointer can also be used to create strings. Pointer variables of **char** type are treated as string.

char \*str = "Hello";

This creates a string and stores its address in the pointer variable **str**. The pointer **str** now points to the first character of the string "Hello". Another important thing to note that string created using **char** pointer can be assigned a value at **runtime**.

char \*str;

str = "hello"; *//thi is Legal*

The content of the string can be printed using printf() and puts().

printf("%s", str);

puts(str);

Notice that **str** is pointer to the string, it is also name of the string. Therefore we do not need to use indirection operator \*.

**Array of Pointers**

We can also have array of pointers. Pointers are very helpful in handling character array with rows of varying length.

char \*name[3]={

"Adam",

"chris",

"Deniel"

};

*//Now see same array without using pointer*

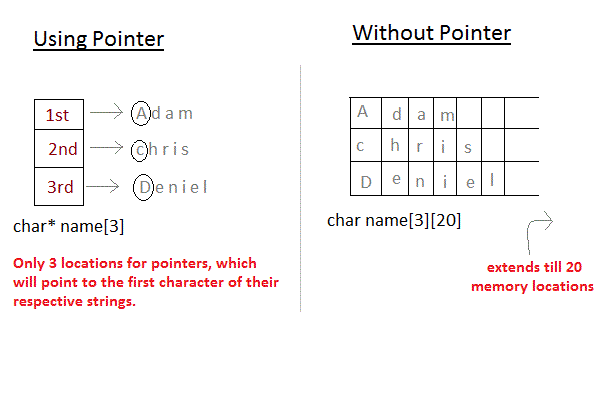
char name[3][20]= {

"Adam",

"chris",

"Deniel"

};



**Pointer to Structure**

Like we have array of integers, array of pointer etc, we can also have array of structure variables. And to make the use of array of structure variables efficient, we use **pointers of structure type**. We can also have pointer to a single structure variable, but it is mostly used with array of structure variables.

struct **Book**

{

char name[10];

int price;

}

int main()

{

struct **Book** a; *//Single structure variable*

struct Book\* ptr; *//Pointer of Structure type*

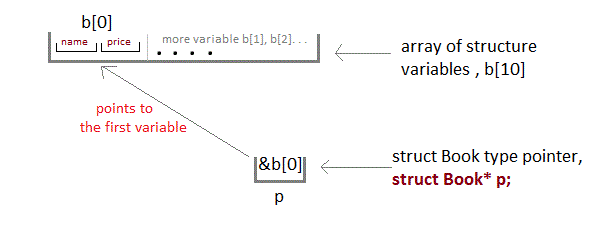
ptr = &a;

struct **Book** b[10]; *//Array of structure variables*

struct Book\* p; *//Pointer of Structure type*

p = &b;

}



**Accessing Structure Members with Pointer**

To access members of structure with structure variable, we used the dot . operator. But when we have a pointer of structure type, we use arrow -> to access structure members.

struct **Book**

{

char name[10];

int price;

}

int main()

{

struct **Book** b;

struct Book\* ptr = &b;

ptr->name = "Dan Brown"; *//Accessing Structure Members*

ptr->price = 500;

}

**Pointer as Function parameter**

Pointer in function parameter list is used to hold address of argument passed during function call. This is also known as **call by reference**. When a function is called by reference any change made to the reference variable will effect the original variable.

**Example: Sorting an array using Pointer**

#include <stdio.h>

#include <conio.h>

void sorting(int \*x, int y);

void main()

{

int a[5],b,c;

clrscr();

printf("enter 5 nos");

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

{

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

}

sorting(a, 5);

getch();

}

void sorting(int \*x, int y)

{

int i,j,temp;

for(i=1; i<=y-1; i++)

{

for(j=0; j\*(x+j+1))

{

temp=\*(x+j);

\*(x+j)=\*(x+j+1);

\*(x+j+1)=temp;

}

}

}

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

{

printf("\t%d",\*(x+i));

}

}

**Function returning Pointer**

A function can also return a pointer to the calling function. In this case you must be careful, because local variables of function doesn't live outside the function, hence if you return a pointer connected to a local variable, that pointer be will pointing to nothing when function ends.

#include <stdio.h>

#include <conio.h>

int\* larger(int\*, int\*);

void main()

{

int a=15;

int b=92;

int \*p;

p=larger(&a, &b);

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

}

int\* larger(int \*x, int \*y)

{

if(\*x > \*y)

return x;

else

return y;

}

**Safe ways to return a valid Pointer.**

1. Either use **argument with functions**. Because argument passed to the functions are declared inside the calling function, hence they will live outside the function called.
2. Or, use **static local variables** inside the function and return it. As static variables have a lifetime until main() exits, they will be available througout the program.

**Pointer to functions**

It is possible to declare a pointer pointing to a function which can then be used as an argument in another function. A pointer to a function is declared as follows,

*type* (**\*pointer-name**)(*parameter*);

**Example :**

int (\*sum)(); //legal declaraction of pointer to function

int \*sum(); //This is not a declaraction of pointer to function

A function pointer can point to a specific function when it is assigned the name of the function.

int sum(int, int);

int (\*s)(int, int);

s = sum;

**s** is a pointer to a function **sum**. Now **sum** can be called using function pointer **s** with the list of parameter.

s (10, 20);

**Example of Pointer to Function**

#include <stdio.h>

#include <conio.h>

int sum(int x, int y)

{

return x+y;

}

int main( )

{

int (\*fp)(int, int);

fp = sum;

int s = fp(10, 15);

printf("Sum is %d",s);

getch();

return 0;

}

Output : 25

**Complicated Function Pointer Example**

You will find a lot of complex function pointer examples around, lets see one such example and try to understand it.

void \*(\*foo) (int\*) ;

It appears complex but it is very simple. In this case **(\*foo)** is a pointer to the function, whose return type is**void\*** and argument is of **int\*** type.

**File Handling in C Language**

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. It is a ready made structure.

In C language, we use a structure **pointer of file type** to declare a file.

FILE **\*fp**;

C provides a number of functions that helps to perform basic file operations. Following are the functions,

|  |  |
| --- | --- |
| **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 begining point |

**Opening a File or Creating a File**

The fopen() function is used to create a new file or to open an existing file.

**General Syntax :**

\*fp = FILE **\*fopen**(const char *\*filename*, const char *\*mode*);

Here **filename** is the name of the file to be opened and **mode** specifies the purpose of opening the file. Mode can be of following types,

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

|  |  |
| --- | --- |
| **mode** | **description** |
| r | opens a text file in reading mode |
| w | opens or create a text file in writing mode. |
| a | opens a text file in append mode |
| r+ | opens a text file in both reading and writing mode |
| w+ | opens a text file in both reading and writing mode |
| a+ | opens a text file in both reading and writing mode |
| rb | opens a binary file in reading mode |
| wb | opens or create a binary file in writing mode |
| ab | opens a binary file in append mode |
| rb+ | opens a binary file in both reading and writing mode |
| wb+ | opens a binary file in both reading and writing mode |
| ab+ | opens a binary file in both reading and writing mode |

**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**.

**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**()) != EOF)

printf("%c",ch);

fclose(fp);

}

**Reading and Writing from File using fprintf() and fscanf()**

#include<stdio.h>

#include<conio.h>

struct emp

{

char name[10];

int age;

};

void main()

{

struct emp e;

FILE \*p,\*q;

p = **fopen**("*one.txt*", "*a*");

q = **fopen**("*one.txt*", "*r*");

printf("Enter Name and Age");

scanf("%s %d", e.name, &e.age);

**fprintf**(p,"%s %d", e.name, e.age);

fclose(p);

do

{

**fscanf**(q,"%s %d", e.name, e.age);

printf("%s %d", e.name, e.age);

}

while( !feof(q) );

getch();

}

In this program, we have create two FILE pointers and both are refering to the same file but in different modes. **fprintf()** function directly writes into the file, while **fscanf()** reads from the file, which can then be printed on console usinf standard **printf()** function.

**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.

**Reading and Writing in a Binary File**

A Binary file is similar to the text file, but it contains only large numerical data. The Opening modes are mentioned in the table for opening modes above.

**fread()** and **fwrite()** functions are used to read and write is a binary file.

fwrite(data-element-to-be-written, size\_of\_elements,

number\_of\_elements, pointer-to-file);

**fread()** is also used in the same way, with the same arguments like fwrite() function. Below mentioned is a simple example of writing into a binary file

const char **\*mytext** = "The quick brown fox jumps over the lazy dog";

FILE \*bfp= **fopen**("*test.txt*", "*wb*");

if (bfp) {

**fwrite**(*mytext*, *sizeof(char)*, *strlen(mytext)*, *bfp*) ;

fclose(bfp) ;

}

**fseek(), ftell() and rewind() functions**

* **fseek()** - It is used to move the reading control to different positions using fseek function.
* **ftell()** - It tells the byte location of current position of cursor in file pointer.
* **rewind()** - It moves the control to beginning of the file.

**Some File Handling Program Examples**

* [Create a File and Store Information in it](http://www.studytonight.com/c/program-to-write-in-file.php)
* [List all the Files present in a Directory](http://www.studytonight.com/c/program-to-list-files-in-directory.php)
* [Finding Size of a File](http://www.studytonight.com/c/program-to-find-size-of-file.php)
* [Copy Content of one File into Another File](http://www.studytonight.com/c/program-copy-file-to-another-file.php)
* [Reverse the Content of File and Print it](http://www.studytonight.com/c/program-to-reverse-content-of-file.php)

**Error Handling**

C language does not provide direct support for error handling. However few method and variable defined in**error.h** header file can be used to point out error using return value of the function call. In C language, a function return -1 or NULL value in case of any error and a global variable **errno** is set with the error code. So the return value can be used to check error while programming.

C language uses the following functions to represent error

* **perror()** return string pass to it along with the textual represention of current errno value.
* **strerror()** is defined in **string.h** library. This method returns a pointer to the string representation of the current errno value.

**Example**

#include <stdio.h>

#include <errno.h>

#include <stdlib.h>

#include <string.h>

extern int errno;

main( )

{

char \*ptr = malloc( 1000000000UL); *//requesting to allocate 1gb memory space*

if ( ptr == NULL ) //if memory not available, it will return null

{

puts("malloc failed");

puts(strerror(errno));

exit(EXIT\_FAILURE); //exit status failure

}

else

{

free( ptr);

exit(EXIT\_SUCCESS); *//exit status Success*

}

}

Here exit function is used to indicate exit status. Its always a good practice to exit a program with a exit status.**EXIT\_SUCCESS** and **EXIT\_FAILURE** are two macro used to show exit status. In case of program coming out after a successful operation **EXIT\_SUCCESS** is used to show successfull exit. It is defined as 0. **EXIT\_Failure**is used in case of any failure in the program. It is defined as -1.

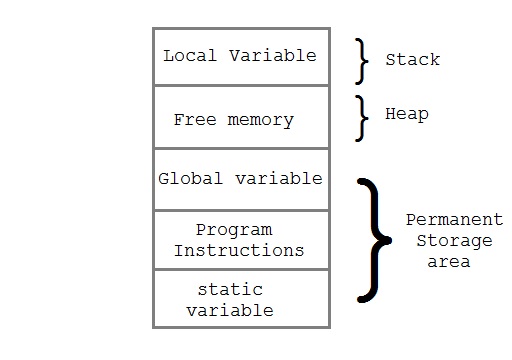
**Dynamic Memory Allocation**

The process of allocating memory at runtime is known as **dynamic memory allocation**. Library routines known as "memory management functions" are used for allocating and freeing memory during execution of a program. These functions are defined in **stdlib.h**.

|  |  |
| --- | --- |
| **Function** | **Description** |
| malloc() | allocates requested size of bytes and returns a void pointer pointing to the first byte of the allocated space |
| calloc() | allocates space for an array of elements, initialize them to zero and then return a void pointer to the memory |
| free | releases previously allocated memory |
| realloc | modify the size of previously allocated space |

**Memory Allocation Process**

**Global** variables, **static** variables and program instructions get their memory in **permanent** storage area whereas **local** variables are stored in area called **Stack**. The memory space between these two region is known as **Heap** area. This region is used for dynamic memory allocation during execution of the program. The size of heap keep changing.



**Allocating block of Memory**

**malloc()** function is used for allocating block of memory at runtime. This function reserves a block of memory of given size and returns a pointer of type void. This means that we can assign it to any type of pointer using typecasting. If it fails to locate enough space it returns a NULL pointer.

**Example using malloc() :**

int \*x;

x = (int\*)malloc(50 \* sizeof(int)); *//memory space allocated to variable x*

free(x); *//releases the memory allocated to variable x*

**calloc()** is another memory allocation function that is used for allocating memory at runtime. **calloc** function is normally used for allocating memory to derived data types such as **arrays** and **structures**. If it fails to locate enough space it returns a NULL pointer.

**Example using calloc() :**

struct employee

{

char \*name;

int salary;

};

typedef struct employee emp;

emp \*e1;

e1 = (emp\*)calloc(30,sizeof(emp));

**realloc()** changes memory size that is already allocated to a variable.

**Example using realloc() :**

int \*x;

x=(int\*)malloc(50 \* sizeof(int));

x=(int\*)realloc(x,100); *//allocated a new memory to variable* ***x***

**Diffrence between malloc() and calloc()**

| **calloc()** | **malloc()** |
| --- | --- |
| calloc() initializes the allocated memory with 0 value. | malloc() initializes the allocated memory with garbage values. |
| Number of arguments is 2 | Number of argument is 1 |
| **Syntax :**  (cast\_type \*)calloc(blocks , size\_of\_block); | **Syntax :**  (cast\_type \*)malloc(Size\_in\_bytes); |

**Command Line Argument**

Command line argument is a parameter supplied to the program when it is invoked. Command line argument is an important concept in C programming. It is mostly used when you need to control your program from outside. command line arguments are passed to **main()** method.

**Syntax :**

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

Here **argc** counts the number of arguments on the command line and **argv[ ]** is a pointer array which holds pointers of type char which points to the arguments passed to the program.

**Example for Command Line Argument**

#include <stdio.h>

#include <conio.h>

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

{

int i;

if( argc >= 2 )

{

printf("The arguments supplied are:\n");

for(i=1;i< argc;i++)

{

printf("%s\t",argv[i]);

}

}

else

{

printf("argument list is empty.\n");

}

getch();

return 0;

}

Remember that **argv[0]** holds the name of the program and **argv[1]** points to the first command line argument and argv[n] gives the last argument. If no argument is supplied, argc will be one.

## Program to find Factorial of a Number

Following is the program to find factorial of a number using for loop.

#include<stdio.h>

#include<conio.h>

void main()

{

int fact,i,n;

fact = 1;

printf("Enter the number\t");

scanf("%d" , &n);

for(i = 1; i <= n; i++)

{

**fact** = **fact\*i;**

}

printf("Factorial of %d is %d", n , fact);

getch();

}

**Output**

Enter the number 5

Factorial of 5 is 120

### Program to find Factorial of a Number using Recursion

In this program we will not use for loop, but we will find the factorial using recursion.

#include<stdio.h>

#include<conio.h>

int factorial(int n);

void main()

{

int fact, i, n;

printf("Enter the number\t");

scanf("%d", &n);

fact = **factorial**(*n*);

printf("Factorial of %d is %d", n, fact);

getch();

}

int **factorial**(*int n*)

{

int fact = 1;

if(n==1)

{

return fact;

}

else

{

fact = **n \* factorial(n-1)**;

return fact;

}

}

**Output**

Enter the number 5

Factorial of 5 is 120

## Program to Reverse a String

Following is the program to Reverse a String using for loop.

#include<stdio.h>

#include<conio.h>

void main()

{

int i, j, k;

char str[100];

char rev[100];

printf("Enter a string\t");

scanf("%s", str);

printf("The original string is %s", str);

for(i = 0; str[i] != '\0'; i++);

{

k = i-1;

}

for(j = 0; j <= i-1; j++)

{

**rev[j] = str[k]**;

k--;

}

printf("The reverse string is %s", rev);

getch();

}

**Output**

Enter a string studytonight

The original string is studytonight

The reverse string is thginotyduts

### Program to Reverse a String using Recursion

Now let's reverse a string using recursion.

#include<stdio.h>

#include<conio.h>

char\* **reverse**(*char\* str*);

void main()

{

int i, j, k;

char str[100];

char \*rev;

printf("Enter a string\t");

scanf("%s", str);

printf("The original string is %s", str);

rev = **reverse**(*str*);

printf("The reverse string is %s", rev);

getch();

}

char\* **reverse**(*char \*str*)

{

static int i=0;

static char rev[100];

if(\*str)

{

reverse(str+1);

rev[i++] = \*str;

}

return rev;

}

**Output**

Enter a string studytonight

The original string is studytonight

The reverse string is thginotyduts

## Program to print Fibonacci Series

Following is the program to print Fibonacci series using while loop.

#include<stdio.h>

#include<conio.h>

void **fibonacci**(int num);

void main()

{

int num = 0;

clrscr();

printf("Enter a number\t");

scanf("%d", &num);

**fibonacci**(*num*);

getch();

}

void fibonacci(int num)

{

int a, b, c, i;

a = 0;

b = 1;

i = 1;

printf("%d\t%d", a, b);

while(i <= num)

{

c = a+b;

a = b;

b = c;

printf("\t%d", b);

i++;

}

}

**Output**

Enter a number 5

0 1 1 2 3 5 8

#### Program to print Fibonacci Series using Recursion

Now lets print the fibonacci series using recursion.

#include<stdio.h>

#include<conio.h>

void **fibonacci**(*int num*);

void main()

{

int n;

int i = 0, j = 1;

printf("Enter the range of the Fibonacci series: ");

scanf("%d", &n);

printf("Fibonacci Series: ");

printf("%d %d ", i, j);

**fibonacci**(*n*);

getch();

}

void fibonacci(int n){

static long int first = 0, second = 1, sum;

if(n > 0)

{

sum = first + second;

first = second;

second = sum;

printf("%ld ", sum);

**fibonacci**(*n-1*);

}

}

**Output**

Enter the range of the Fibonacci series: 5

Fibonacci Series: 0 1 1 2 3 5 8

## Program to find sum of digits of a Number

Following is the program to find out sum of digits of a number using loops.

#include<stdio.h>

#include<conio.h>

int **sumOfDigit**(*int num*);

void main()

{

int num, sum;

clrscr();

printf("Enter a number\t");

scanf("%d", &num);

sum = **sumOfDigit**(*num*);

printf("The sum of digit %d is %d", num, sum);

getch();

}

int **sumOfDigit**(*int num*)

{

int s=0, a, r;

a = num;

while(a)

{

r = a%10;

s = s+r;

a = a/10;

}

return s;

}

**Output**

Enter a number 123

The sum of digit 123 is 6

### Program to find sum of digits of a Number using recursion

Following is the same program using recursion.

#include<stdio.h>

#include<conio.h>

int **sumOfDigit**(int num);

void main()

{

int num, sum;

clrscr();

printf("Enter a number\t");

scanf("%d", &num);

sum = **sumOfDigit**(*num*);

printf("The sum of digit %d is %d", num, sum);

getch();

}

int **sumOfDigit**(*int num*)

{

int s, a;

s = s+(num%10);

a = num/10;

if(a > 0)

{

**sumOfDigit**(a);

}

return s;

}

**Output**

Enter a number 108

The sum of digit 108 is 9

**Program to Sort an Array element**

#include<stdio.h>

#include<conio.h>

void **sorting**(int \*x,int y);

void main()

{

int a[5],b,c;

clrscr();

printf("Enter 5 numbers");

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

{

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

}

**sorting**(a,5);

getch();

}

void **sorting**(int \*x,int y)

{

int i,j,temp;

for(i = 1; i <= y-1; i++)

{

for(j = 0; j < y-i; j++)

{

if(\*(x+j) >\*(x+j+1))

{

temp = \*(x+j);

\*(x+j) = \*(x+j+1);

\*(x+j+1) = temp;

}

}

}

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

{

printf("\t%d",\*(x+i));

}

}

**Output**

Enter 5 numbers 26 19 21 7 12

7 12 19 21 26

## Program to Swap Variable's Numeric Values

In this program we will swap values of two variables. The values are numeric and we will swap them using different techniques(some tricky ones too). Let's start with the simplest technique, which is by using a**temporary variable**

#include<stdio.h>

#include<conio.h>

void main()

{

int x=10, y=15, temp;

temp = x;

x = y;

y = temp;

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

getch();

}

**Output**

x= 15 y= 10

### Program to Swap two Numbers without using Third variable

#include<stdio.h>

#include<conio.h>

void main()

{

int x=10, y=15;

x = **x+y-(y=x)**;

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

getch();

}

**Output**

x= 15 y= 10

### Program to Swap two Numbers using Bitwise Operator

#include<stdio.h>

#include<conio.h>

void main()

{

int x=6, y=4;

x = x^y;

y = x^y;

x = x^y;

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

getch();

}

**Output**

x= 4 y= 6

### Program to Swap two Numbers using Division and Multiplication

#include<stdio.h>

#include<conio.h>

void main()

{

int x=6, y=4;

x = x\*y;

y = x/y;

x = x/y;

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

getch();

}

**Output**

x= 4 y= 6

**Program to Find the Largest Element of an Array using Recursion**

You can do this in many ways, Let's learn how to do this using recursion.

#include<stdio.h>

#include<conio.h>

int **findLargest**(int *arr[]*,int *size*);

void main()

{

int arr[5];

int i, max=0;

clrscr();

printf("Enter 5 numbers\t");

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

{

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

}

max = **findLargest**(arr, 5);

printf("The largest element is %d", max);

getch();

}

int **findLargest**(int *\*arr*,int *size*)

{

static int i=0, max=-999;

if(i < size)

{

if( max < \*(arr+i) )

{

max = \*(arr+i);

}

i++;

**findLargest**(arr, size);

}

return max;

}

**Output**

Enter 5 numbers 209 194 185 167 198

The largest element is 209

**Program to Check whether a Number is Pallindrome**

#include<stdio.h>

#include<conio.h>

void main()

{

int a, b, c, s=0;

clrscr();

printf("Enter a number:\t");

scanf("%d", &a);

c = a;

*//the number is reversed inside the while loop.*

while(a > 0)

{

b = a%10;

s = (s\*10)+b;

a = a/10;

}

*//Here the reversed number is compared with the given number*

if(s == c)

{

printf("The number %d is a pallindrome", c);

}

else {

printf("The number %d is not a pallindrome", c);

}

getch();

}

**Output**

Enter a numbers 121

The number 121 is a pallindrome

**Program to Check whether a Number is Pallindrome or not using Recursion**

#include<stdio.h>

#include<conio.h>

void **checkPallindrome**(int *num*);

static int p;

void main()

{

int num;

clrscr();

printf("Enter a number:\t");

scanf("%d", &num);

p = num;

**checkPallindrome**(*num*);

getch();

}

int b;

static int s = 0;

void **checkPallindrome**(int num)

{

static int c;

c = num;

b = num%10;

s = (s\*10)+b;

num = num/10;

if(num>0)

{

**checkPallindrome**(*num*);

}

else

{

if(s == p)

{

printf("The number %d is a pallindrome", p);

}

else {

printf("The number %d is not a pallindrome", p);

}

}

}

**Output**

Enter a numbers 143

The number 143 is not a pallindrome

**Program to Remove duplicate Element in an Array**

Below is a simple and easy to understand program to find and remove any duplicate element present in specified array.

#include<stdio.h>

#include<conio.h>

void main()

{

int a[20], i, j, k, n;

clrscr();

printf("\nEnter array size : ");

scanf("%d",&n);

printf("\nEnter %d array element : ", n);

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

{

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

}

printf("\nOriginal array is : ");

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

{

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

}

printf("\nNew array is : ");

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

{

for(j=i+1; j < n; )

{

if(a[j] == a[i])

{

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

{

a[k] = a[k+1];

}

n--;

}

else {

j++;

}

}

}

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

{

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

}

getch();

}

**Output**

Enter array size : 5

Enter 5 array element : 11 13 11 12 13

Original array is : 11 13 11 12 13

New array is : 11 13 12

## Program to Create a File & Write Data in it

In this program we will be creating a new file and will then store information in it.

#include<stdio.h>

#include<conio.h>

void main()

{

FILE \*fptr;

char name[20];

int age;

float salary;

*/\* open for writing \*/*

fptr = fopen("emp.txt", "w");

if (fptr == NULL)

{

printf("File does not exists \n");

return;

}

printf("Enter the name \n");

scanf("%s", name);

fprintf(fptr, "Name = %s\n", name);

printf("Enter the age\n");

scanf("%d", &age);

fprintf(fptr, "Age = %d\n", age);

printf("Enter the salary\n");

scanf("%f", &salary);

fprintf(fptr, "Salary = %.2f\n", salary);

fclose(fptr);

}

You can add any information in the file, like we have added Name, Age and Salary for some employees, you can change the program as per your requirements.

**Program to Print names of all Files present in a Directory**

#include<stdio.h>

#include<dirent.h>

int main(void)

{

DIR \*d;

struct **dirent** \*dir;

d = opendir(".");

if (d)

{

while ((dir = **readdir**(d)) != NULL)

{

printf("%s\n", dir->d\_name);

}

closedir(d);

}

return(0);

}

**Output**

File1.txt

File2.txt

File3.txt

File4.txt

File5.txt

File6.txt

File7.txt

## Program to Find Size of a File

We will be using fseek() and ftell() functions to find the size of the file. There are others ways to find the file size, like looping on the whole content of file and finding out the size, but using File Handling functions makes it easier.

#include<stdio.h>

#include<conio.h>

void main()

{

FILE \*fp;

char ch;

int size = 0;

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

if (fp == NULL)

{

printf("\nFile unable to open ");

}

else

{

printf("\nFile opened ");

}

fseek(fp, 0, 2); /\* file pointer at the end of file \*/

size = ftell(fp); /\* take a position of file pointer un size variable \*/

printf("The size of given file is : %d\n", size);

fclose(fp);

}

**Program to Copy Content of One File into Another File**

We already know how to open a file, read contents of a file and write into a file. So in this program, we will read from one file and simultaneously write into the other file, till we reach end of first file.

#include<stdio.h>

void main()

{

*/\**

*File\_1.txt is the file with content and,*

*File\_2.txt is the file in which content of File\_1*

*will be copied.*

*\*/*

FILE \*fp1, \*fp2;

char ch;

int pos;

if ((fp1 = fopen("File\_1.txt","r")) == NULL)

{

printf("\nFile cannot be opened");

return;

}

else

{

printf("\nFile opened for copy...\n ");

}

fp2 = fopen("File\_2.txt", "w");

**fseek**(fp1, 0L, SEEK\_END); *// file pointer at end of file*

pos = **ftell**(fp1);

fseek(fp1, 0L, SEEK\_SET); *// file pointer set at start*

while (pos--)

{

ch = **fgetc**(fp1); *// copying file character by character*

**fputc**(ch, fp2);

}

fcloseall();

}

**Program to Reverse the Contents of a File and Print it**

#include<stdio.h>

#include<errno.h>

*// to count the total number of characters inside the source file*

long **count\_characters**(FILE \*);

void main()

{

int i;

long cnt;

char ch, ch1;

FILE \*fp1, \*fp2;

if (fp1 = fopen("*File\_1.txt*", "*r*"))

{

printf("The FILE has been opened...\n");

fp2 = fopen("*File\_2.txt*", "*w*");

cnt = count\_characters(fp1);

*/\**

*makes the pointer fp1 to point at the*

*last character of the file*

*\*/*

**fseek**(fp1, -1L, 2);

printf("Number of characters to be copied %d\n", **ftell**(fp1));

while (*cnt*)

{

ch = fgetc(fp1);

fputc(ch, fp2);

fseek(fp1, -2L, 1); *// shifts the pointer to the previous character*

cnt--;

}

printf("\n\*\*File copied successfully in reverse order\*\*\n");

}

else

{

perror("Error occured\n");

}

fclose(fp1);

fclose(fp2);

}

*/\**

*count the total number of characters in the file*

*that \*f points to*

*\*/*

long **count\_characters**(FILE \*f)

{

fseek(f, -1L, 2);

long last\_pos = **ftell**(f); *//returns the position of the last element of the file*

last\_pos++;

return last\_pos;

}