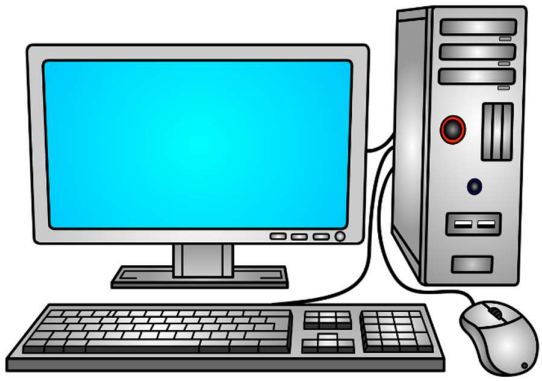
C Programming

Syllabus

Introduction to Programming

A computer is a programmable electronic device that accepts raw data as input and processes it with a set of instructions (a program) to produce the result as output. It renders output just after performing mathematical and logical operations and can save the output for future use. It can process numerical as well as non-numerical calculations. The term "computer" is derived from the Latin word "computare" which means to calculate.



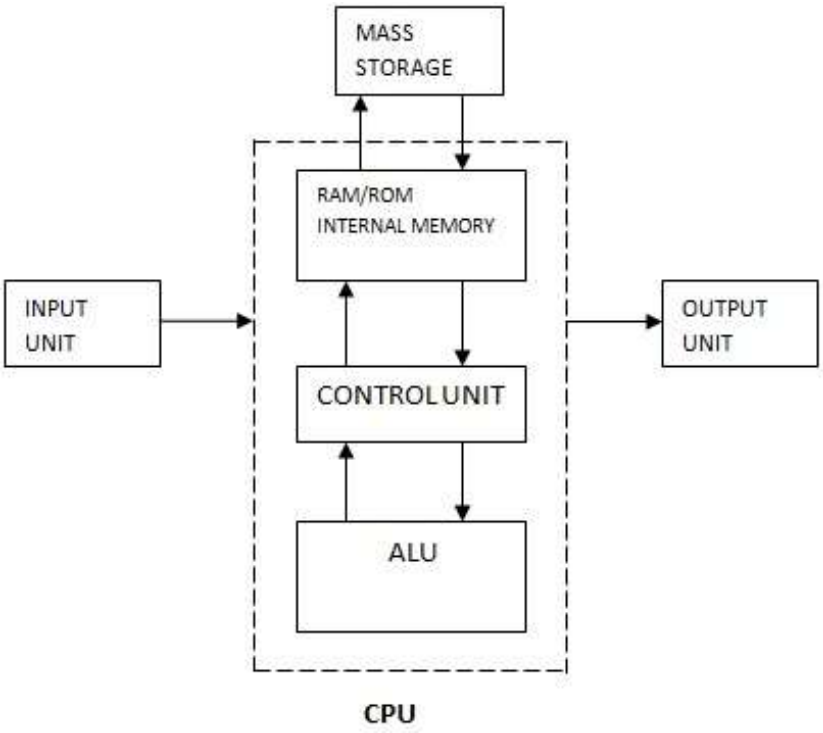
A computer program is the process that professionals use to write code that instructs how a computer, application or software program performs. At its most basic, computer programming is a set of instructions to facilitate specific actions.

The basic components of a computer are:

1. Input unit
2. Central Processing Unit(CPU)
3. Output unit

The CPU is further divided into three parts-

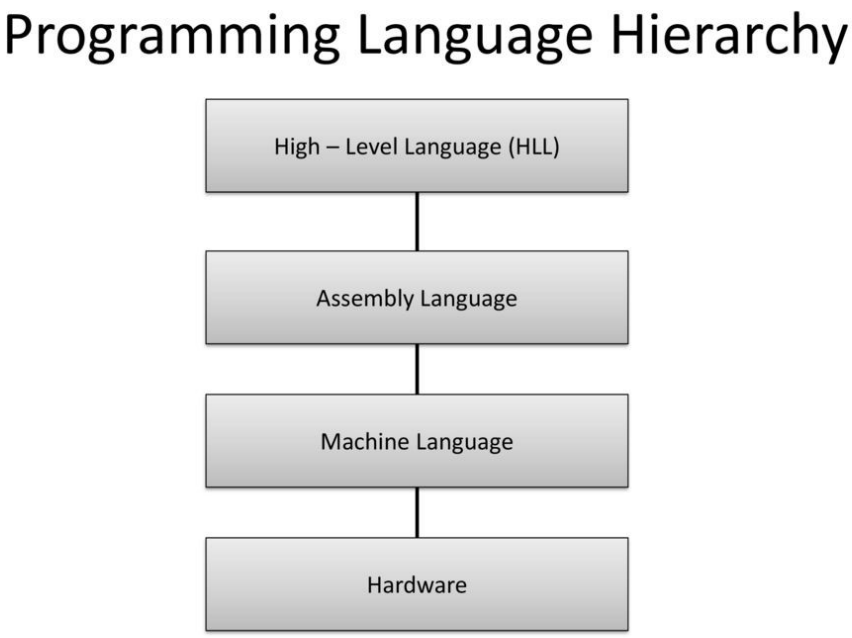
- Memory unit
- Control unit
- Arithmetic Logic unit



A **High-Level Language** (HLL) is a programming language such as C, FORTRAN, or Pascal that enables a programmer to write programs that are more or less independent of a particular type of computer. Such languages are considered high-level because they are closer to human languages and further from machine languages

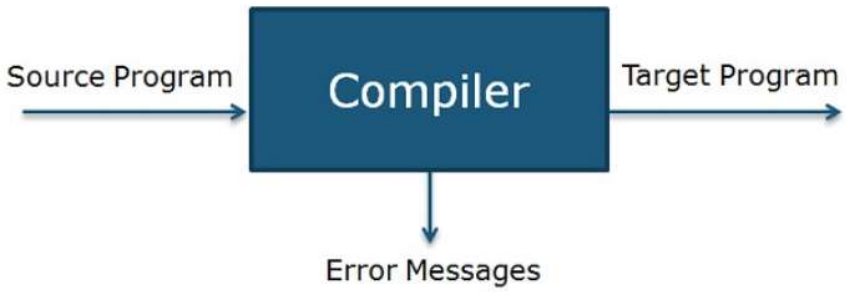
A **Low-Level Language** (LLL) is a programming language that provides little or no abstraction of programming concepts and is very close to writing actual machine instructions. Two examples of low-level languages are assembly and machine code.

An **Assembly language** is a low-level programming **language** designed for a specific type of processor. It may be produced by compiling source **code** from a high-level programming **language** (such as C/C++) but can also be written from scratch.



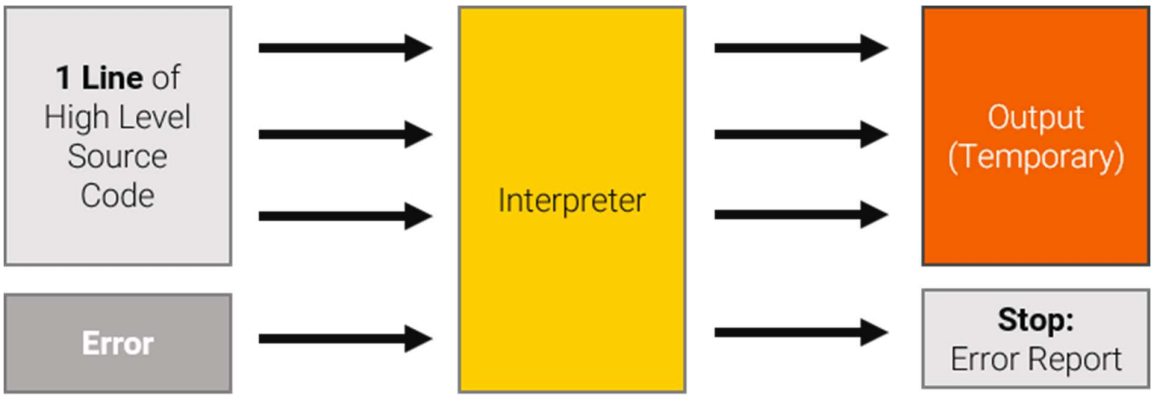
A **compiler** is a program that reads a program written in the high-level language and converts it into the machine or low-level language and reports the errors present in the program.

It converts the entire source code in one go or could take multiple passes to do so, but at last, the user gets the compiled code which is ready to execute. The returned **target code** file can be run with many different inputs, over and over. the compiler doesn't need to be around for any subsequent reruns.

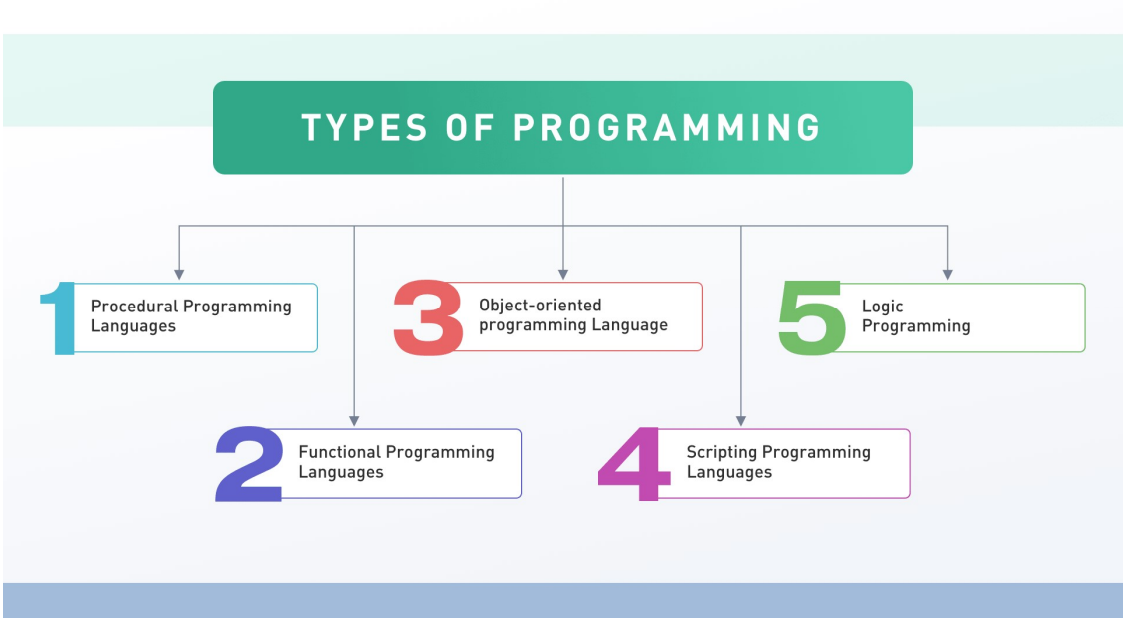


An **interpreter** is a computer program that is used to directly execute program instructions written using one of the many high-level programming languages.

The interpreter transforms the high-level program into an intermediate language that it then executes, or it could parse the high-level source code and then performs the commands directly, which is done line by line or statement by statement.



Types of Programming Languages



Procedural Programming Language

The procedural programming language is used to execute a sequence of statements which lead to a result. Typically, this type of programming language uses multiple variables, heavy loops and other elements, which separates them from functional programming languages. Functions of procedural language may control variables, other than function's value returns. For example, printing out information.

Functional Programming Language

Functional programming language typically uses stored data, frequently avoiding loops in favor of recursive functions.The functional programing's primary focus is on the return values of functions, and side effects and different suggests that storing state are powerfully discouraged. For example, in an exceedingly pure useful language, if a

function is termed, it's expected that the function not modify or perform any o/p.

Object-oriented Programming Language

This programming language views the world as a group of objects that have internal data and external accessing parts of that data. The aim this programming language is to think about the fault by separating it into a collection of objects that offer services which can be used to solve a specific problem. One of the main principle of object oriented programming language is encapsulation that everything an object will need must be inside of the object. This language also emphasizes reusability through inheritance and the capacity to spread current implementations without having to change a great deal of code by using polymorphism.

Scripting Programming Language

These programming languages are often procedural and may comprise object-oriented language elements, but they fall into their own category as they are normally not full-fledged programming languages with support for development of large systems. For example, they may not have compile-time type checking. Usually, these languages require tiny syntax to get started.

Logic Programming Language

These types of languages let programmers make declarative statements and then allow the machine to reason about the consequences of those statements. In a sense, this language doesn't tell the computer how to do something, but employing restrictions on what it must consider doing.

To call these groups ” types of language ” is really a bit confusing. It’s easy to program in an object-oriented style in C language. In truth, most of the languages include ideas and features from various domains, which only helps to increase the usefulness of these types of languages. Nevertheless, most of the programming languages do not best in all styles of programming.

Introduction To C Programming

In 1972, a great computer scientist Dennis Ritchie created a new programming language called 'C' at the Bell Laboratories. It was created from 'ALGOL', 'BCPL' and 'B' programming languages. 'C' programming language contains all the features of these languages and many more additional concepts that make it unique from other languages.

Some Facts About C Programming Language

- In **1988**, the **American National Standards Institute (ANSI)** had formalized the C language.
- C was invented to write **UNIX** operating system.
- C is a successor of 'Basic Combined Programming Language' (BCPL) called **B language.**
- **Linux OS**, **PHP**, and **MySQL** are written in C.
- C has been written in assembly language

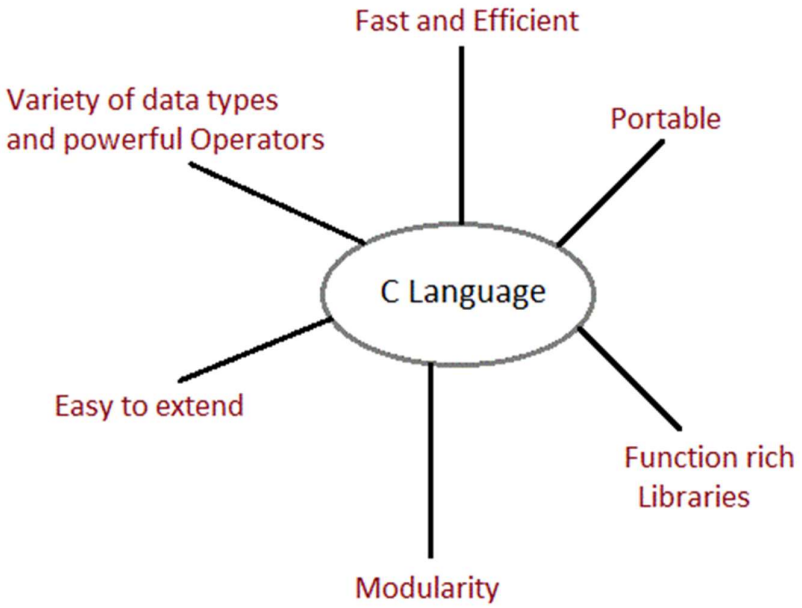
Reasons For Popularity of C Language

- Easy to learn
- Structured language
- It produces efficient programs.
- It can handle low-level activities.
- It can be compiled on a variety of computers.

Advantages of C

- C is the building block for many other programming languages.
- Programs written in C are highly portable.

- Several standard functions are there (like in-built) that can be used to develop programs.
- C programs are collections of C library functions, and it's also easy to add functions to the C library.
- The modular structure makes code debugging, maintenance, and testing easier.



Disadvantages Of C

- C does not provide Object Oriented Programming (OOP) concepts.
- There are no concepts of Namespace in C.
- C does not provide binding or wrapping up of data in a single unit.
- C does not provide Constructor and Destructor.

Structure of C Program

Header File	**# include <stdio.h>**
Main( )	**int main( )** **{**
Variable Declaration	**int a=10;**
Body	**printf("%d",a);**
Return	**return 0;** **}**

The components of the above structure are:

1. **Header Files Inclusion**: The first component is to include the Header files in a C program.
   A header file is a file with extension .h which contains C function declarations and macro definitions to be shared between several source files.

Some of C Header files:

- **stddef.h** – Defines several useful types and macros.
- **stdint.h** – Defines exact width integer types.
- **stdio.h** – Defines core input and output functions
- **stdlib.h** – Defines numeric conversion functions, pseudo-random network generator, memory allocation
- **string.h** – Defines string handling functions
- **math.h** – Defines common mathematical functions

Syntax to include a header file in C:
#include

****

2. **Main Method Declaration:** The next part of a C program is to declare the main() function. The syntax to declare the main function is:

Syntax to Declare main method:
int main()

{}

****

3. **Variable Declaration:** It refers to the variables that are to be used in the function. In a C program, the variables are to be declared before any operation in the function.

Example:
int main()

{**int a;**

****

4. **Body:** Body of a function in C program, refers to the operations that are performed in the functions. It can be anything like manipulations, searching, sorting, printing, etc.

Example:
int main()

{

int a;

printf("%d", a);

****

5. **Return Statement:**The return statement refers to the returning of the values from a function. This return statement and return value depend upon the return type of the function. For example, if the return type is void, then there will be no return statement.
   Example:

```
int main()
{
 int a;
printf("%d", a);
return 0;
}
```

Lets write our 1st program in C Language

```
#include <stdio.h>
int main(void)
{
        printf("TechnoXamm");
        return 0;
}
/* Program Execution Completed */
```

Line 1: [#include <stdio.h>] In a C program, all lines that start with # are processed by preprocessor . In a very basic

term,preprocessor takes a C program and produces another C program. The produced program has no lines starting with #, all such lines are processed by the preprocessor. In the above example, preprocessor copies the preprocessed code of stdio.h to our file. The .h files are called header files in C. These header files generally contain declaration of functions.

Line 2 [int main(void)] There must to be starting point from where execution of compiled C program begins. In C, the execution typically begins with first line of main(). The void written in brackets indicates that the main doesn't take any parameter. The int written before main indicates return type of main().

Line 3 and 6: [{ and }] In C language, a pair of curly brackets define a scope and mainly used in functions and control statements like if, else, loops. All functions must start and end with curly brackets.

Line 4 [printf("TechnoXamm");] printf() is a standard library function to print something on standard output. The semicolon at the end of printf indicates line termination. In C, semicolon is always used to indicate end of statement.

Line 5 [return 0;] The return statement returns the value from main(). The returned value may be used by operating system to know termination status of your program. The value 0 typically means successful termination.

Line 7 [/* */]These are comment lines or in other words these line of code are not executable by the compiler.

Introduction to IDE

An IDE, or Integrated Development Environment, enables programmers to consolidate the different aspects of writing a computer program.

IDEs increase programmer productivity by combining common activities of writing software into a single application: editing source code, building executables, and debugging.

Features of using an IDE

- **Editing Source Code**

  Writing code is an important part of programming. We start with a blank file, write a few lines of code, and a program is born! IDEs facilitate this process with features like syntax highlighting and autocomplete.

- **Syntax Highlighting**

An IDE that knows the syntax of your language can provide visual cues. Keywords, words that have special meaning like class in Java, are highlighted with different colors.

- **Autocomplete**

  When the IDE knows your programming language, it can anticipate what you're going to type next!.We've seen statements with printf() quite a bit so far. In an IDE, we might see printf() as an autocomplete option after only typing pr. This saves keystrokes so the programmer can focus on logic in their code.

- **Debugging**

  No programmer avoids writing bugs and programs with errors.When a program does not run correctly, IDEs provide debugging tools that allow programmers to examine different variables and inspect their code in a deliberate way.

  IDEs also provide hints while coding to prevent errors **before** compilation.

Elements of C Programming Language

As every language has some basic geometrical rules and elements, similarly C language has some elements and rules for building a program which has some meaning.

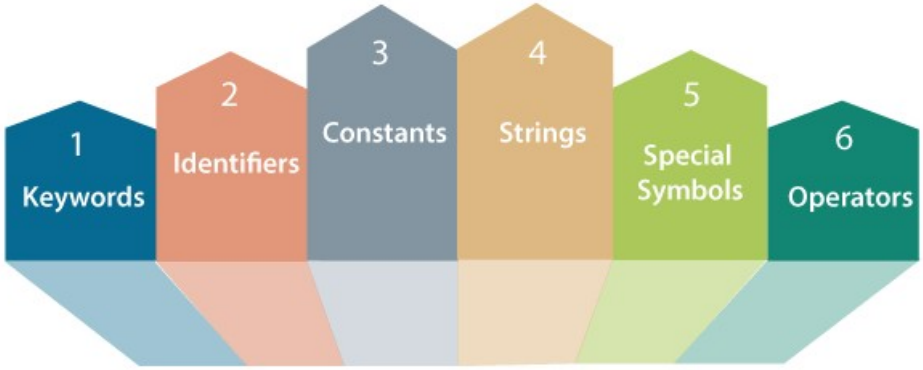
Character Set: In Real world to communicate with people we use language like Hindi English Urdu extra which is constructed and Defined by some characters, words extra. Similarly in C programming language we have various characters to communicate with the computer in order to produce a meaningful program and can produce an output.

Alphabets	A, B, ..., Y, Z a, b, ..., y, z
Digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Special symbols	~ ' ! @ # % ^ & * ( ) _ - + = | \ { } [ ] : ; " ' < > , . ? /

Tokens in C

We can define the token as the smallest individual element in C. For `example, we cannot create a sentence without using words; similarly, we cannot create a program in C without using tokens in C.

Therefore, we can say that tokens in C is the building block or the basic component for creating a program in C language.



Classification of C Tokens

Keywords:

1) they are those elements of C language whose meaning has already being defined or explained and has seeds task.

2) keyword cannot be used to assign new meaning to the keywords.

Auto	do	goto	signed	unsigned	break
void	else	int	case	static	double
sizeof	enum	long	struct	char	if
while	const	extern	register	continue	volatile
defualt	for	typeodef	float	short	return
union	const				

Identifiers:

An identifier is nothing but a name assigned to an element in a program. Example, name of a variable, function, etc. Identifiers are the user-defined names consisting of 'C' standard character set.

Following rules must be followed for identifiers:

1. The first character must always be an alphabet or an underscore.
2. It should be formed using only letters, numbers, or underscore.
3. A keyword cannot be used as an identifier.
4. It should not contain any whitespace character.
5. The name must be meaningful.

Constants:

A constant is a value assigned to the variable which will remain the same throughout the program, i.e., the constant value cannot be changed.

There are two ways of declaring constant:

- Using const keyword
- Using #define pre-processor

Constant	Example
Integer constant	**10, 11, 34, etc.**
Floating-point constant	**45.6, 67.8, 11.2, etc.**
Octal constant	**011, 088, 022, etc.**
Hexadecimal constant	**0x1a, 0x4b, 0x6b, etc.**
Character constant	**'a', 'b', 'c', etc.**
String constant	**"java", "c++", ".net", etc.**

Strings:

Strings in C are always represented as an array of characters having null character '\0' at the end of the string. This null character denotes the end of the string. Strings in C are enclosed within double quotes, while characters are enclosed within single characters. The size of a string is a number of characters that the string contains.

Strings in different ways:

char a[10] = "TechnoXamm"; // The compiler allocates the 10 bytes to the 'a' array.

char a[] = " TechnoXamm "; // The compiler allocates the memory at the run time.

char a[10] = {'t','e','c','h','n','o','x','a','m','m','\0'}; // String is represented in the form of characters.

Special Characters

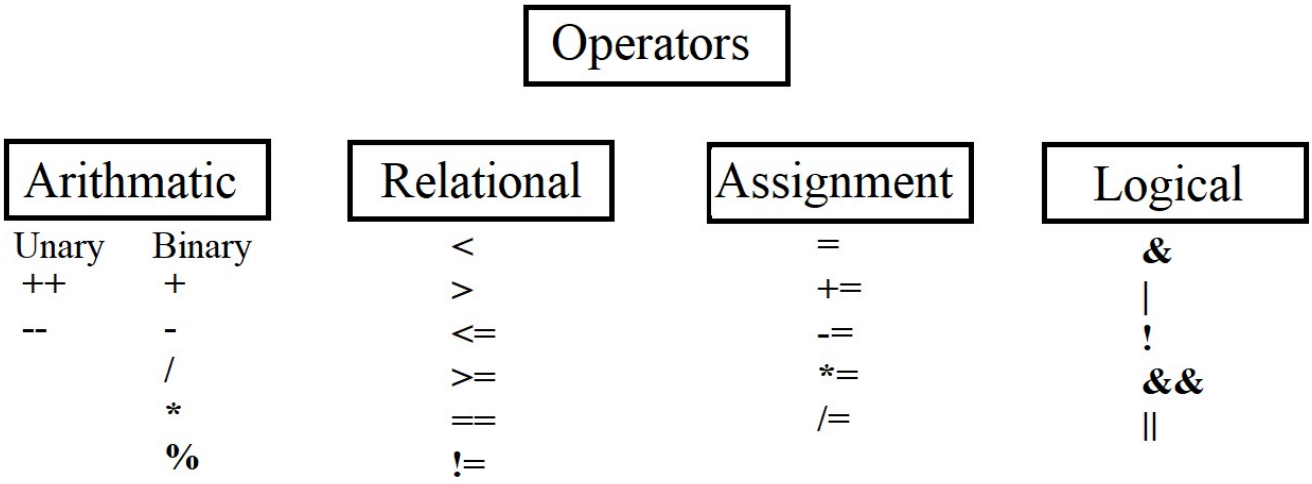
Some special characters are used in C, and they have a special meaning which cannot be used for another purpose.

- **Square brackets [ ]:** The opening and closing brackets represent the single and multidimensional subscripts.
- **Simple brackets ( ):** It is used in function declaration and function calling. For example, printf() is a pre-defined function.
- **Curly braces { }:** It is used in the opening and closing of the code. It is used in the opening and closing of the loops.
- **Comma (,):** It is used for separating for more than one statement and for example, separating function parameters in a function call,

- separating the variable when printing the value of more than one variable using a single printf statement.
- **Hash/pre-processor (#):** It is used for pre-processor directive. It basically denotes that we are using the header file.
- **Asterisk (*):** This symbol is used to represent pointers and also used as an operator for multiplication.
- **Tilde (~):** It is used as a destructor to free memory.
- **Period (.):** It is used to access a member of a structure or a union.

Operators

Operators in C is a special symbol used to perform the functions. The data items on which the operators are applied are known as operands. Operators are applied between the operands. Depending on the number of operands, operators are classified as follows:



Comments

A comment is an explanation or description of the source code of the program. It helps a developer explain logic of the code and improves program readability. At run-time, a comment is ignored by the compiler.

There are two types of comments in C:

1) A comment that starts with a slash asterisk /* and finishes with an asterisk slash */ and you can place it anywhere in your code, on the same line or several lines.

2) Single-line Comments which uses a double slash // dedicated to comment single lines

Example of single line comment :

```
// A single line comment
```

Example of Multi Line comment :

```
/* Multiline

   Comment */
```

Operators in C Language

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

There are following types of operators to perform different types of operations in C language.

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Ternary or Conditional Operators
- Assignment Operator

Precedence of Operators in C

The precedence of operator species that which operator will be evaluated first and next. The associativity specifies the operator direction to be evaluated; it may be left to right or right to left.
Example:

```
int value = 10 + 20 * 10
```

The value variable will contain **210** because * (multiplicative operator) is evaluated before + (additive operator).

The precedence and associativity of C operators is given below:

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

Arithmetic Operators

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

Arithmetic Operators are used to performing mathematical calculations like addition (+), subtraction (-), multiplication (*), division (/) and modulus (%).

Relational Operators

Relational operators are used to comparing two quantities or values.

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 Operator

C language supports following 3 logical operators : -

- AND
- OR

- NOT

Lets understand this with an example. 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

Operator	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
<<	left shift

>>	right shift

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

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** operator, 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 have to be shifted. Both operands have the same precedence.

Conditional Operators

The conditional operators in C language are known by two more names

1. **Ternary Operator**
2. **? : Operator**

It is actually the if condition that we use in C language decision making, but using conditional operator, we turn the if condition statement into a short and simple operator.

The syntax of a conditional operator is :

expression 1 ? expression 2 : expression 3

Explanation:

- The question mark **"?"** in the syntax represents the **if** part.
- The first expression (expression 1) generally returns either true or false, based on which it is decided whether (expression 2) will be executed or (expression 3)
- If (expression 1) returns true then the expression on the left side of **" : "** i.e (expression 2) is executed.
- If (expression 1) returns false then the expression on the right side of **" : "** i.e (expression 3) is executed.

Assignment Operator

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

Control Statements

In C, the control flows from one instruction to the next instruction until now in all programs. This control flow from one command to the next is called sequential control flow. Nonetheless, in most C programs the programmer may want to skip instructions or repeat a set of instructions repeatedly when writing logic. This can be referred to as sequential control flow. The declarations in C let programmers make such decisions which are called decision-making or control declarations.

C also supports an unconditional set of branching statements that transfer the control to another location in the program. Selection declarations in C.

1. If statements
2. Switch Statement
3. Conditional Operator Statement
4. Goto Statement

If Statements

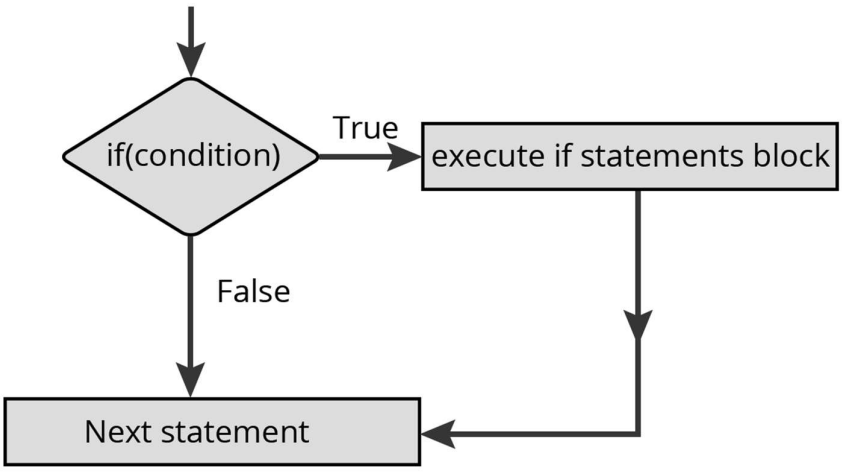
If statement enables the programmer to choose a set of instructions, based on a condition. When the condition is evaluated to true, a set of instructions will be executed and a different set of instructions will be executed when the condition is evaluated to false. We have 4 types of if Statement which are:

I. If Statement

The condition evaluates to either true or false. True is always a non-zero value, and false is a value that contains zero. Instructions can be a single instruction or a code block enclosed by curly braces { }.

Syntax :

```
if (condition)
{
    Statement ;
}
```



Example:

```
#include<stdio.h>
int main()
{
        int num1=1;
        int num2=2;
        if(num1<num2)              //test-condition
        {
                printf("num1 is smaller than num2");
        }
        return 0;
}
```

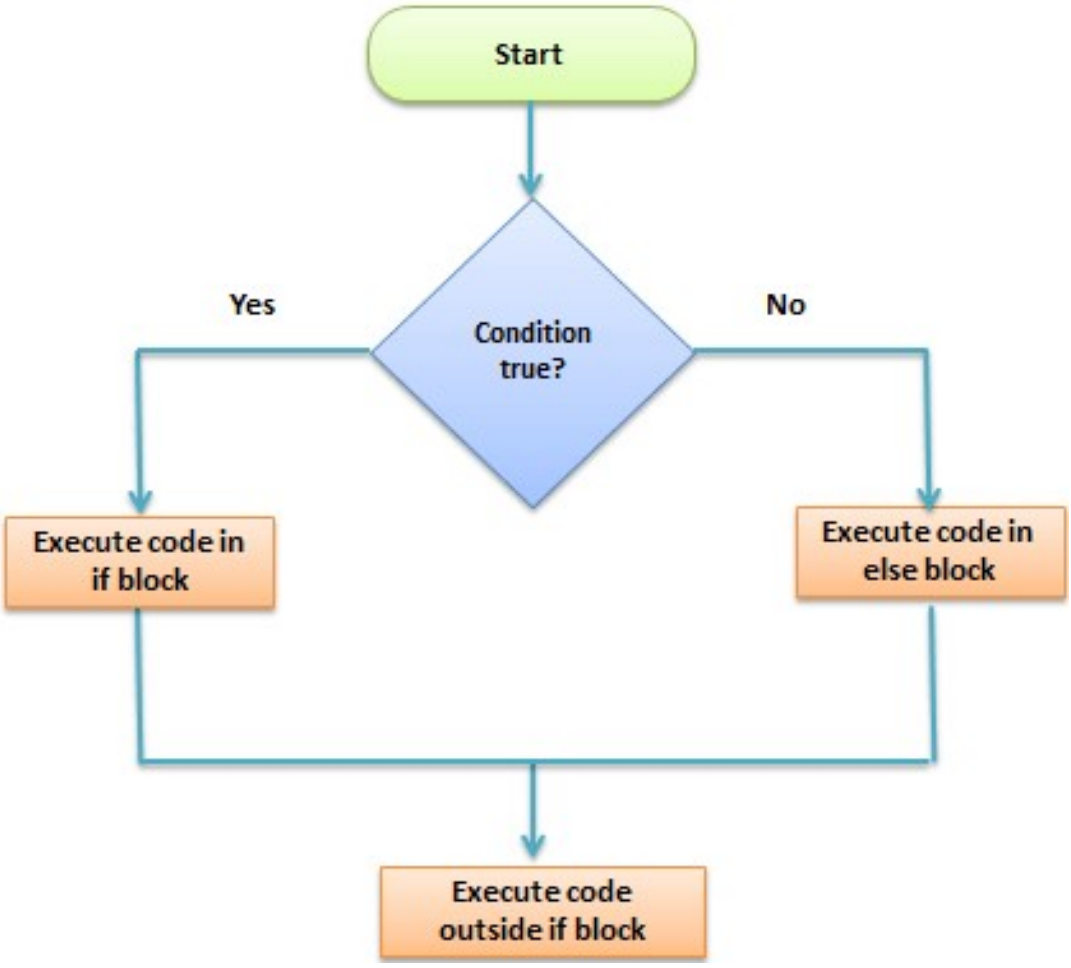
```
num1 is smaller than num2
```

II. If – else Statement

In this statement, there are two types of statements execute. First, if the condition is true first statement will execute if the condition is false second condition will be executed.

Syntax:

```
If(condition)
{
Statement(s);
}
else
{
Statement(s)
}
Statement
```



Example:

```
#include<stdio.h>
int main()
{
	int num=19;
	if(num<10)
	{
		printf("The value is less than 10");
	}
	else
	{
		printf("The value is greater than 10");
	}
	return 0;
}
```

```
The value is greater than 10
```

III. Else -if

The else..if statement is useful when you need to check multiple conditions within the program, nesting of if-else blocks can be avoided using else..if statement.

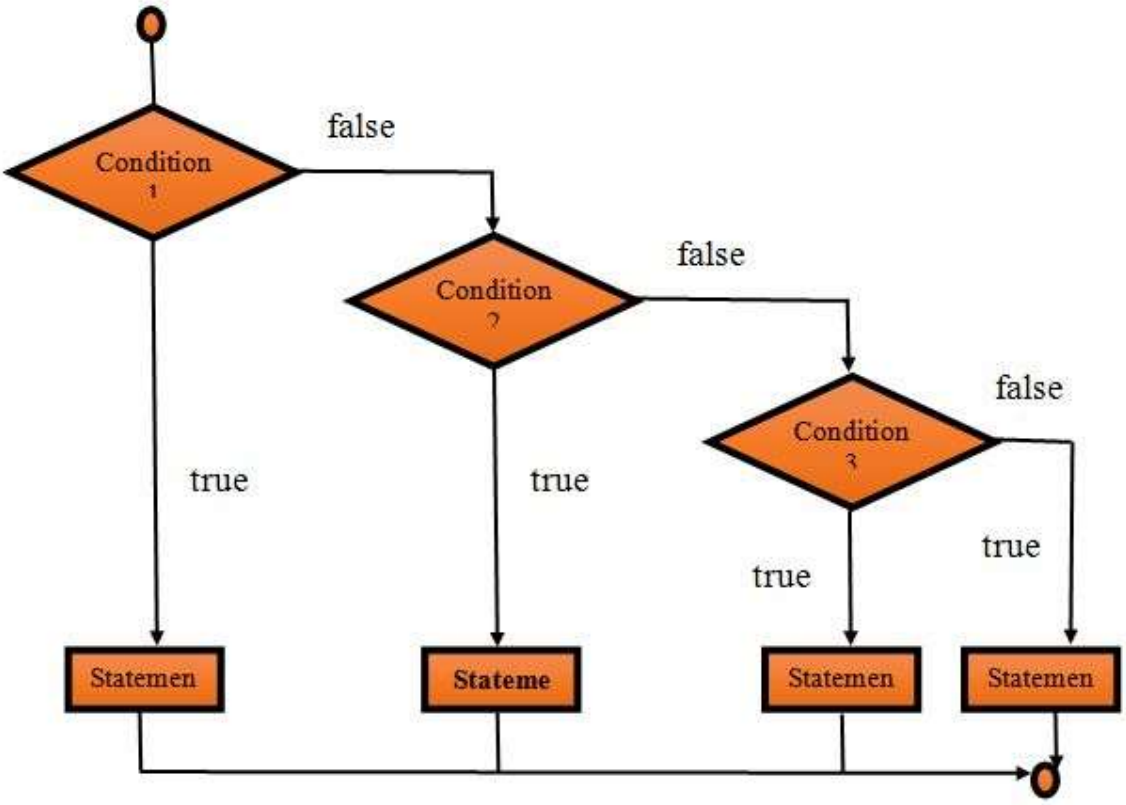
Syntax:

```
if (condition1)
{
 //These statements would execute if the condition1 is true
}
 else if(condition2)
{
 //These statements would execute if the condition2 is true
}
 else if (condition3)
{
 //These statements would execute if the condition3 is true
}
else
{
  //These statements would execute if all the conditions return false
}
```

Example:

```
#include<stdio.h>

main()
{
 int a, b;

 printf("Please enter the value for a:");
 scanf("%d", & amp; a);

 printf("\nPlease enter the value for b:");
 scanf("%d", & amp; b);

 if (a & gt; b)
 {
  printf("\n a is greater than b");
 }
 else if (b & gt; a)
 {
  printf("\n b is greater than a");
 }
 else
 {
  printf("\n Both are equal");
 }
}
```

```
Please enter value of a:5
Please enter value of b:7
b is greater than a
```



IV. Nested If – Else

When a series of decision is required, nested if-else is used. Nesting means using one if-else construct within another one.

When an if else statement is present inside the body of another “if” or “else” then this is called nested if else.

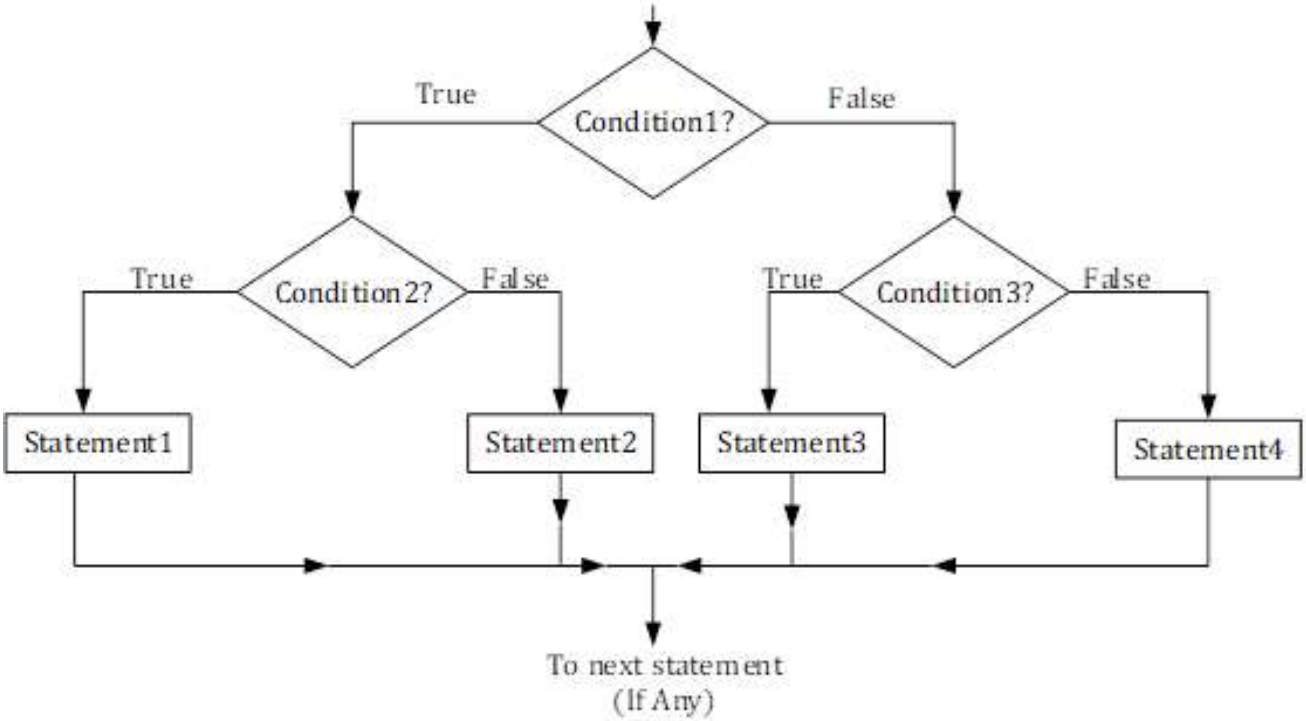
Syntax:

```
if(condition)
{
  //Nested if else inside the body of "if"
    if(condition2)
 {
    //Statements inside the body of nested "if"
  }
    else
{
    //Statements inside the body of nested "else"
  }
}
   else
{
  //Statements inside the body of "else"
}
```

Example:

```
#include<stdio.h>
int main()
{
	int num=1;
	if(num<10)
	{
		if(num==1)
		{
			printf("The value is:%d\n",num);
		}
		else
		{
			printf("The value is greater than 1");
		}
	}
	else
	{
		printf("The value is greater than 10");
	}
	return 0;
}
```

```
The value is:1
```



Switch Statements

Switch statement in C tests the value of a variable and compares it with multiple cases. Once the case match is found, a block of statements associated with that particular case is executed.

Each case in a block of a switch has a different name/number which is referred to as an identifier. The value provided by the user is compared with all the cases inside the switch block until the match is found.

Syntax:

```
switch( expression )
{
    case value-1:
            Block-1;
            break;
    case value-2:
            Block-2;
            break;
    case value-n:
            Block-n;
            break;
    default:
            Block-1;
            break;
}
```

Rules for using switch statement

1. The expression (after switch keyword) must yield an integer value i.e the expression should be an integer or a variable or an expression that evaluates to an integer.
2. The case label values must be unique.

3. The case label must end with a colon(:)
4. The next line, after the case statement, can be any valid C statement.

Points to Remember

1. We don't use those expressions to evaluate switch case, which may return floating point values or strings or characters.
2. break statements are used to **exit** the switch block. It isn't necessary to use break after each block, but if you do not use it, then all the consecutive blocks of code will get executed after the matching block.
3. **default** case is executed when none of the mentioned case matches the switch expression. The default case can be placed anywhere in the switch case. Even if we don't include the default case, switch statement works.
4. Nesting of switch statements are allowed, which means you can have switch statements inside another switch block.

Difference between switch and if

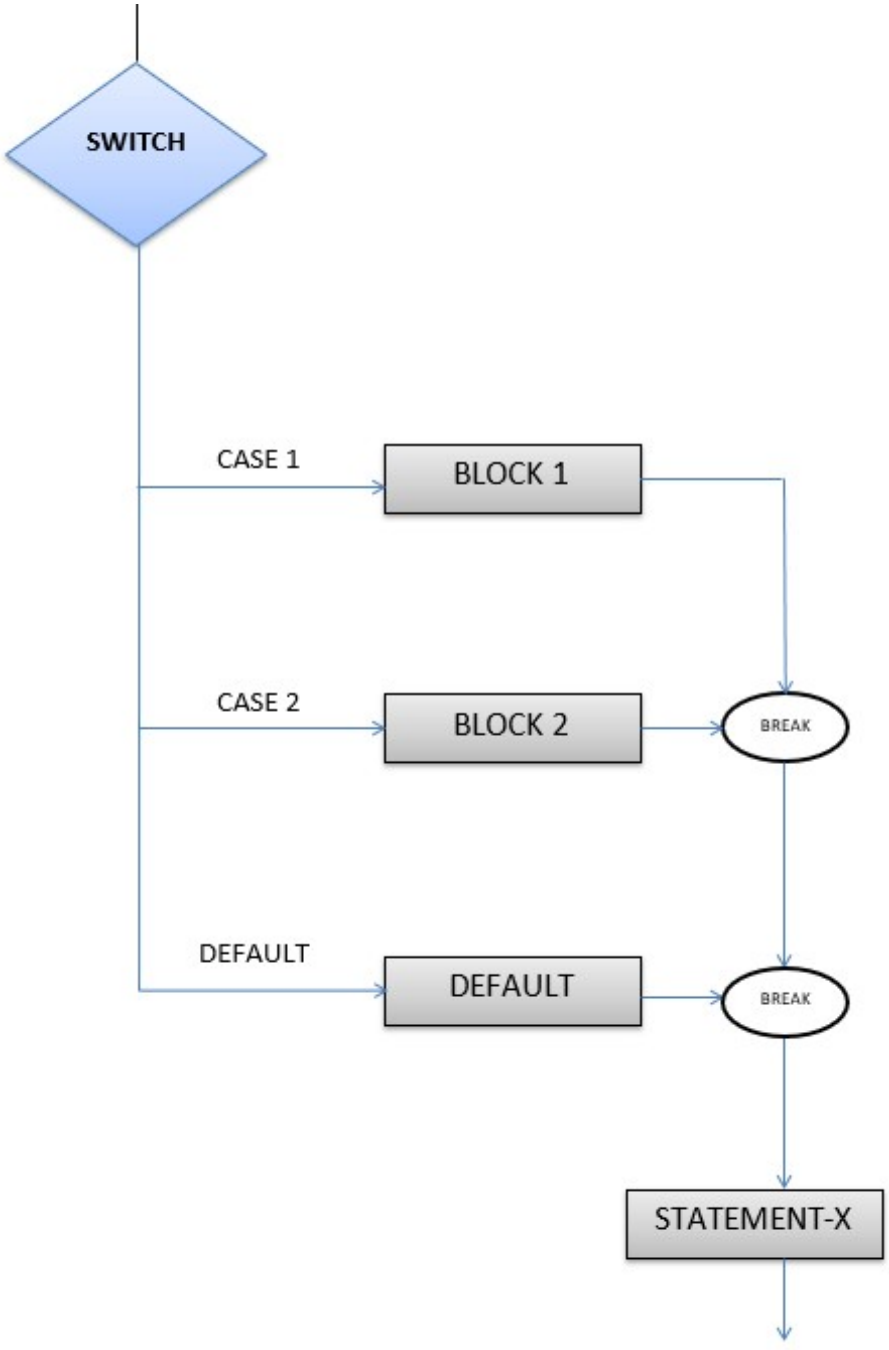
- if statements can evaluate float conditions. switch statements cannot evaluate float conditions.
- if statement can evaluate relational operators. switch statement cannot evaluate relational operators i.e they are not allowed in switch statement.

Example:

```
#include <stdio.h>
int main()
{
    char ch='b';
    switch (ch)
    {
        case 'd':
          printf("CaseD ");
          break;
        case 'b':
          printf("CaseB");
          break;
        case 'c':
          printf("CaseC");
          break;
        case 'z':
          printf("CaseZ ");
          break;
```

```
default:
          printf("Default ");
    }
    return 0;
}
```

CaseB


SWITCH
CASE 1
BLOCK 1
CASE 2
BLOCK 2
BREAK
DEFAULT
DEFAULT
BREAK
STATEMENT-X

Conditional Operator Statement

It is similar to the if-else statement. The if-else statement takes more than one line of the statements, but the conditional operator finishes the same task in a single statement. The conditional operator in C is also called the ternary operator because it operates on three operands.

The operands may be an expression, constants or variables. It starts with a condition, hence it is called a conditional operator.

Syntax:

expression1 ? expression2 : expression3;

or

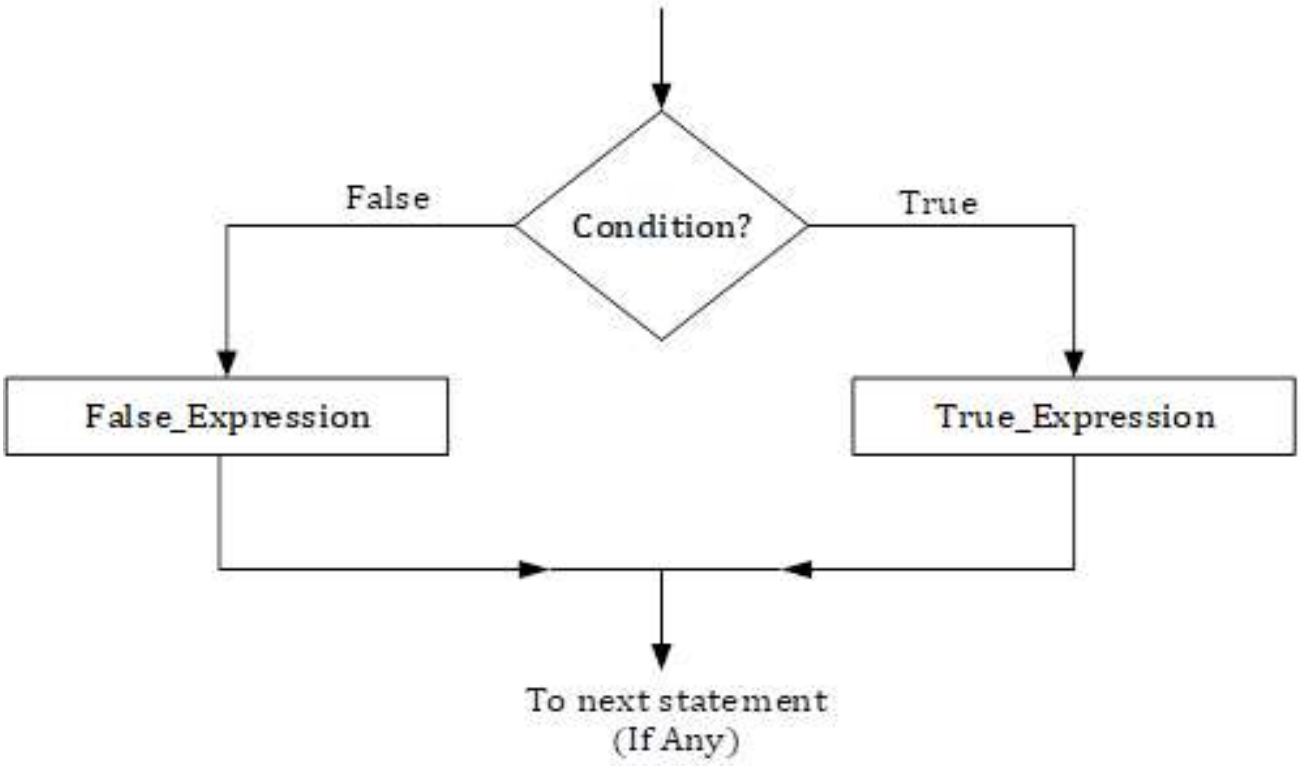
condition ? true statement : false statement;

Example:

```
#include<stdio.h>
int main()
{
  float num1, num2, max;
  printf("Enter two numbers: ");
  scanf("%f %f", &num1, &num2);
```

```
max = (num1 > num2) ? num1 : num2;
printf("Maximum of %.2f and %.2f = %.2f",num1, num2, max);
return 0;
}
```

```
Enter two numbers: 12.5 10.5
Maximum of 12.50 and 10.50 = 12.50
```



Looping Structures

A Loop executes the sequence of statements many times until the stated condition becomes false. A loop consists of two parts, a body of a loop and a control statement. The control statement is a combination of some conditions that direct the body of the loop to execute until the specified condition becomes false. The purpose of the loop is to repeat the same code a number of times.

Depending upon the position of a control statement in a program, looping in C is classified into two types:

1. Entry controlled loop

2. Exit controlled loop

In an **entry controlled loop,** a condition is checked before executing the body of a loop. It is also called as a pre-checking loop.

In an **exit controlled loop**, a condition is checked after executing the body of a loop. It is also called as a post-checking loop.

The control conditions must be well defined and specified otherwise the loop will execute an infinite number of times. The loop that does not stop executing and processes the statements number of times is called as an **infinite loop**. An infinite loop is also called as an "**Endless loop**." Following are some characteristics of an infinite loop:

1. No termination condition is specified.

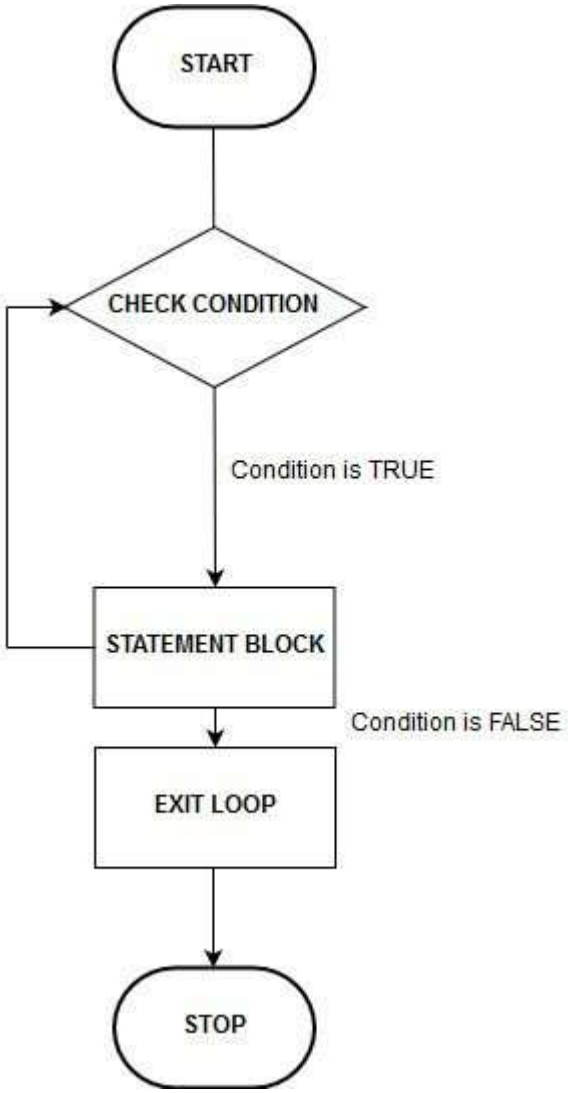
2. The specified conditions never meet.

'C' programming language provides us with three types of loop constructs:

1. The while loop

2. The do-while loop

3. The for loop



I. While Loop :

While loop in C is a pre-test loop where the expression is evaluated then only statements are executed. It uses a test expression to control the loop. Before every iteration of the loop, the test expression is evaluated.

Syntax :

```
   while(condition)
{
   //code
}
```

Example :

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i = 1;
while( i <=10 ) {
printf ("%d " , i ," “);
i++;
}
getch();
}
```

```
1 2 3 4 5 6 7 8 9 10
```

II. Do – While Loop

It also executes the code until condition is false. In this at least once, code is executed whether condition is true or false but this is not the case with while. While loop is executed only when the condition is true.

Syntax:

```
do
{
//code
}while(condition);
```

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i = 1;
do{
printf ("%d " , i ," ");
i++;
}while( i < =10 );
getch();
```

```
1 2 3 4 5 6 7 8 9 10
```

III. For Loop

When you know exactly how many times you want to loop through a block of code, use the for loop instead of a while loop:

It also executes the code until condition is false. In this three parameters are given that is

- Initialization
- Condition
- Increment/Decrement

Syntax :

```
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

or

```
for (Initialization; Condition; Increment/Decrement) {
  // code block to be executed
}
```

Statement 1 is executed (one time) before the execution of the code block.

Statement 2 defines the condition for executing the code block.

Statement 3 is executed (every time) after the code block has been executed.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i;
for( i = 20; i < 25; i++) {
printf ("%d " , i,, “ “);
}
getch();
}
```

```
20 21 22 23 24
```

Nested Loops

A loop inside another loop is called a nested loop. The depth of nested loop depends on the complexity of a problem. We can have any number of nested loops as required. Consider a nested loop where the outer loop runs *n* times and consists of another loop inside it. The inner loop runs *m* times. Then, the total number of times the inner loop runs during the program execution is n*m.

Nested While Loop

Example:

```
#include <stdio.h>

int main()

{

   int i=1,j;

   while (i <= 5)

   {

      j=1;

      while (j <= i )

      {

         printf("%d ",j);

         j++;

      }

printf("n");

        i++;

   }

   return 0;

}
```

```
1

1 2

1 2 3

1 2 3 4

1 2 3 4 5
```

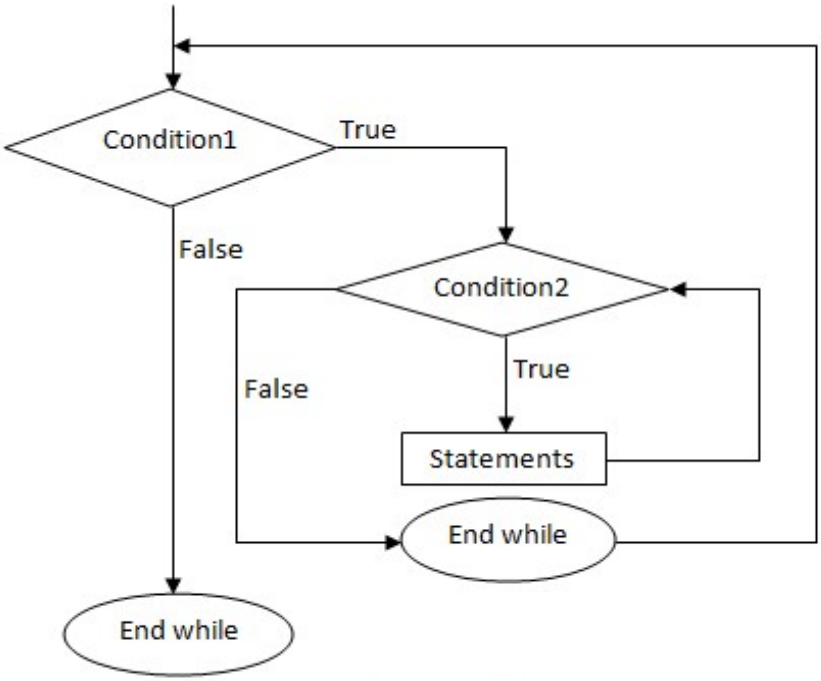


fig: Flowchart for nested while loop

Nested Do – While Loop

```
#include <stdio.h>

int main()

{

   int i=1,j;

   do

   {
```

```
j=1;
        do
        {
            printf("*");
            j++;
        }while(j <= i);
        i++;
        printf("n");
    }while(i <= 5);
    return 0;
}
```

```
*
**
***
****
*****
```

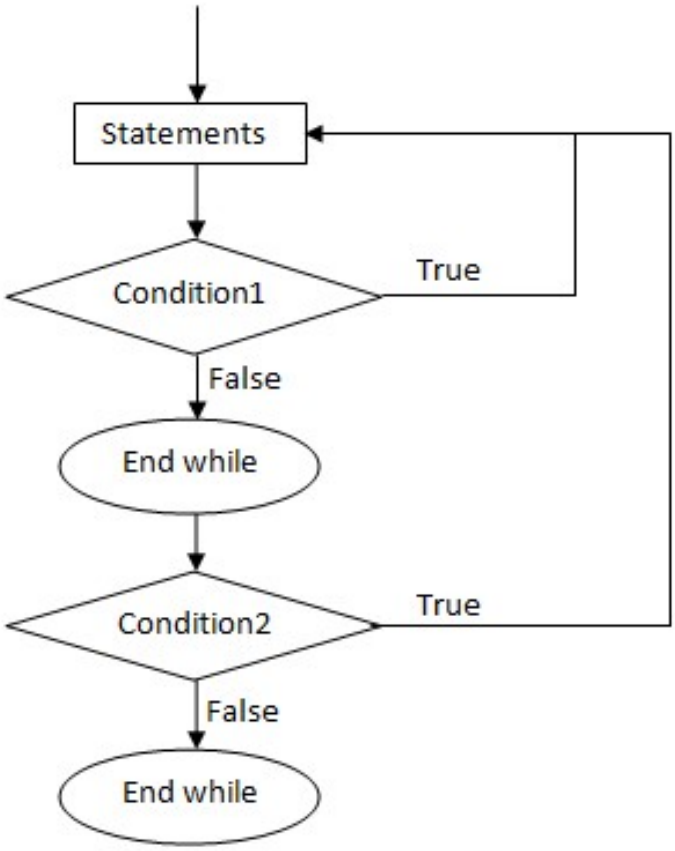


Fig: Flowchart for nested do-while loop

Nested For Loop

```
#include<stdio.h>

#include<math.h>

int main()

{
```

```
int i,j,n;
    printf("Enter a number:");
    scanf("%d",&n);
    for(i=2;i<=n;i++)
    {
        for(j=2;j<=(int)pow(i,0.5);j++)
        {
            if(i%j==0)
            {
                printf("%d is compositen",i);
                break;
            }
        }
    }
    return 0;
}
```

```
Enter a number:15

4 is composite

6 is composite

8 is composite

9 is composite

10 is composite

12 is composite

14 is composite

15 is composite
```

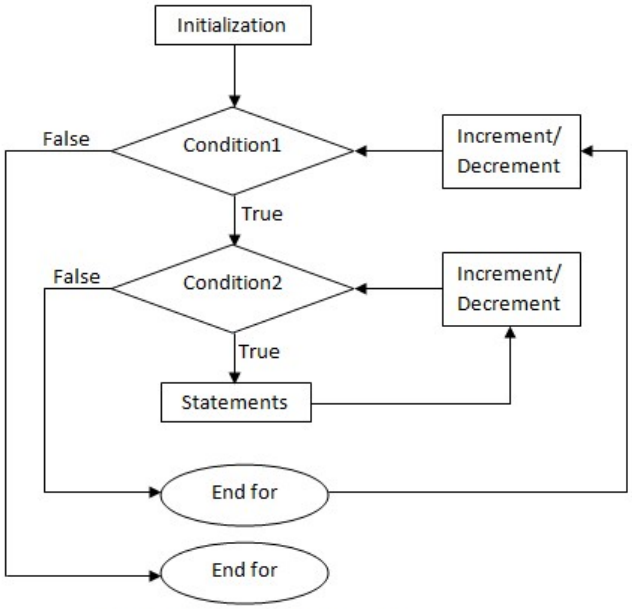


Fig: Flowchart for nested for loop

Jump Statements

Jump Statement makes the control jump to another section of the program unconditionally when encountered. It is usually used to terminate the **loop** or **switch-case** instantly. It is also used to escape the execution of a section of the program.

1. Break Jump Statement

A **break** statement is used to terminate the execution of the rest of the block where it is present and takes the control out of the block to the next statement.

It is mostly used in loops and switch-case to bypass the rest of the statement and take the control to the end of the loop. The use of the **break** keyword in switch-case has been explained in the previous tutorial Switch – Control Statement.

Another point to be taken into consideration is that the **break** statement when used in nested loops only terminates the inner loop where it is used and not any of the outer loops.

```
#include <stdio.h>

int main() {

int i;

for (i = 1; i <= 15; i++) {

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

if (i == 10)

break;
```

```
}

return 0;

}
```

```
1
2
3
4
5
6
7
8
9
10
```

2. Continue Jump Statement

The continue jump statement like any other jump statement interrupts or changes the flow of control during the execution of a program. **Continue** is mostly used in loops.

Rather than terminating the loop it stops the execution of the statements underneath and takes control to the next iteration.

Similar to a break statement, in the case of a nested loop, the continue passes the control to the next iteration of the inner loop where it is present and not to any of the outer loops.

```
#include <stdio.h>
int main() {
int i, j;
for (i = 1; i < 3; i++) {
for (j = 1; j < 5; j++) {
if (j == 2)
continue;
printf("%d\n", j);
}
}
return 0;
}
```

```
1
3
4
1
3
4
```

3. Goto Jump Statement

goto jump statement is used to transfer the flow of control to any part of the program desired. The programmer needs to specify a label or identifier with the goto statement in the following manner:

***goto label*;**

```
#include <stdio.h>
int main() {
int i, j;
for (i = 1; i < 5; i++) {
if (i == 2)
goto there;
printf("%d\n", i);
}
there:
printf("Two");
return 0;
}
```

```
1
Two
```

4. Return Jump Statement

Return jump statement is usually used at the end of a function to end or terminate it with or without a value. It takes the control from the calling function back to the main function(main function itself can also have a **return**).

An important point to be taken into consideration is that **return** can only be used in functions that is declared with a **return** type such as *int, float, double, char,* etc.

The functions declared with void type does not return any value. Also, the function returns the value that belongs to the same data type as it is declared. Here is a simple example to show you how the **return** statement works.

```
#include <stdio.h>
char func(int ascii) {
return ((char) ascii);
}
int main() {
int ascii;
char ch;
printf("Enter any ascii value in decimal: \n");
scanf("%d", & ascii);
ch = func(ascii);
```

```
printf("The character is : %c", ch);
return 0;
}
```

```
Enter any ascii value in decimal:
110
The character is : n
```

Characters in C

As every language contains a set of characters used to construct words, statements, etc., C language also has a set of characters which include **alphabets, digits**, and **special symbols**. C language supports a total of 256 characters.

Every C program contains statements. These statements are constructed using words and these words are constructed using characters from C character set. C language character set contains the following set of characters...

1. Alphabets
2. Digits
3. Special Symbols

Alphabets

C language supports all the alphabets from the English language. Lower and upper case letters together support 52 alphabets.

lower case letters - **a to z**

UPPER CASE LETTERS - **A to Z**

Digits

C language supports 10 digits which are used to construct numerical values in C language.

Digits - **0, 1, 2, 3, 4, 5, 6, 7, 8, 9**

Special Symbols

C language supports a rich set of special symbols that include symbols to perform mathematical operations, to check conditions, white spaces, backspaces, and other special symbols.

Special Symbols - **~ @ # $ % ^ & * ( ) _ - + = { } [ ] ; : ' " / ? . > , < \ | tab newline space NULL bell backspace verticaltab etc.,**

Characters	ASCII Value
A - Z	65 - 90
a - z	97 - 122
0 - 9	48 - 57
Special Symbol	0 - 47, 58 - 64, 91 - 96, 123 - 127

Letter	ASCII Code	Binary	Letter	ASCII Code	Binary
a	097	01100001	A	065	01000001
b	098	01100010	B	066	01000010
c	099	01100011	C	067	01000011
d	100	01100100	D	068	01000100
e	101	01100101	E	069	01000101
f	102	01100110	F	070	01000110
g	103	01100111	G	071	01000111
h	104	01101000	H	072	01001000
i	105	01101001	I	073	01001001
j	106	01101010	J	074	01001010
k	107	01101011	K	075	01001011
l	108	01101100	L	076	01001100
m	109	01101101	M	077	01001101
n	110	01101110	N	078	01001110
o	111	01101111	O	079	01001111
p	112	01110000	P	080	01010000
q	113	01110001	Q	081	01010001
r	114	01110010	R	082	01010010
s	115	01110011	S	083	01010011
t	116	01110100	T	084	01010100
u	117	01110101	U	085	01010101
v	118	01110110	V	086	01010110
w	119	01110111	W	087	01010111
x	120	01111000	X	088	01011000
y	121	01111001	Y	089	01011001
z	122	01111010	Z	090	01011010

Program to print all ASCII Character code

```
#include<stdio.h>
#include<conio.h>
int main() {
        int i;
      clrscr();
        printf("ASCII  ==>  Character\n");
        for(i = -128; i <= 127; i++)
          printf("%d    ==>     %c\n", i, i);
      getch();
        return 0;
}
```

Arrays

In C language, it is a collection of similar type of data which can be either of int, float, double, char (String), etc. All the data types must be same. For example, we can't have an array in which some of the data are integer and some are float.

Array of integers

Suppose we need to store marks of 50 students in a class and calculate the average marks. So, declaring 50 separate variables will do the job but no programmer would like to do so. And there comes **array** in action.

Declaration of an Array

datatype array_name [array_size] ;

For example, take an array of integers 'n'.

int n[6];

n[ ] is used to denote an array 'n'. It means that 'n' is an array.

So, int n[6] means that 'n' is an array of 6 integers. Here, 6 is the **size of the array** i.e. there are 6 elements in the array 'n'.

n

We need to give the size of the array because the complier needs to allocate space in the memory which is not possible without knowing the size. Compiler determines the size required

for an array with the help of the number of elements of an array and the size of the data type present in the array.

Here 'int n[6]' will allocate space to 6 integers.

We can also declare an array by another method.

int n[ ] = {2, 3, 15, 8, 48, 13};

In this case, we are declaring and assigning values to the array at the same time. Here, there is no need to specify the array size because compiler gets it from { 2,3,15,8,48,13 }.

Indexing in an Array

Every element of an array has its index. We access any element of an array using its index.

Pictorial view of the above mentioned array is:

Element	2	3	15	8	48	13
Index	0	1	2	3	4	5

0, 1, 2, 3, 4 and 5 are indices. It is like they are identity of 6 different elements of an array. Index always starts from 0. So, the first element of an array has a index of 0.

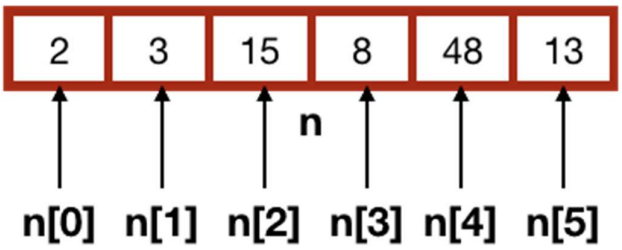
“Index of an array starts with 0.”

We access any element of an array using its index and the syntax to do so is:

array_name[index]

For example, if the name of an array is 'n', then to access the first element (which is at 0 index), we write n[0].

Here,
n[0] is 2
n[1] is 3
n[2] is 15
n[3] is 8
n[4] is 48
n[5] is 13



n[0], n[1], etc. are like any other variables we were using till now i.e., we can set there value as n[0] = 5; like we do with any other variables (x = 5;, y = 6;, etc.).

Assigning Values to Array

By writing int n[ ]={ 2,4,8 }; , we are declaring and assigning values to the array at the same time, thus initializing it.

But when we declare an array like int n[3];, we need to assign values to it separately. Because 'int n[3];' will definitely allocate space of 3 integers in memory but there are no integers in that space.

To initialize it, assign a value to each of the elements of the array.

n[0] = 2;
n[1] = 4;
n[2] = 8;

It is just like we are declaring some variables and then assigning values to them.

int x,y,z;
x=2;
y=4;
z=8;

```
#include <stdio.h>
int main()
{
   int  marks[3];
   float  average;

   printf("Enter marks of first student\n");
   scanf(" %d" , &marks[0]);

   printf("Enter marks of second student\n");
   scanf(" %d" , &marks[1]);

   printf("Enter marks of third student\n");
```

```
scanf(" %d" , &marks[2]);


    average = (marks[0] + marks[1] + marks[2]) / 3.0;
    printf ("Average marks : %f\n" , average);


    return 0;
}
```

```
Enter marks of first student
23
Enter marks of second student
25
Enter marks of third student
30
Average marks : 26.000000
```

In the above example, two points should be kept in mind.
The average value should be of type 'float' because the average of integers can be float also.
Secondly, while taking out the average, the sum of the numbers should be divided by 3.0 and not 3, otherwise you will get the average value as an integer and not float.

```
#include <stdio.h>
int main()
{
   int n[10]; /* declaring n as an array of 10 integers */
   int i,j;
   /* initializing elements of array n */
   for (i = 0; i<10; i++)
   {
      printf("Enter value of n[%d]",i);
      scanf("%d",&n[i]);
   }
   /* printing the values of elements of array */
   for (j = 0; j < 10; j++)
   {
      printf("n[%d] = %d\n", j, n[j]);
   }
   return 0;
}
```

```
Enter value of n[0]12
Enter value of n[1]34
Enter value of n[2]23
Enter value of n[3]78
Enter value of n[4]32
Enter value of n[5]21
Enter value of n[6]4
Enter value of n[7]23
Enter value of n[8]46
Enter value of n[9]24
n[0] = 12
n[1] = 34
n[2] = 23
n[3] = 78
n[4] = 32
n[5] = 21
n[6] = 4
n[7] = 23
n[8] = 46
n[9] = 24
```

Pointer to Arrays

As we all know that pointer is a variable whose value is the address of some other variable i.e., if a variable 'y' points to another variable 'x', it means that the value of the variable 'y' is the address of 'x'.

Similarly, if we say that a variable 'y' points to an array 'n', it would mean that the value of 'y' is the address of the first element of the array i.e. n[0]. It means that the pointer of an array is the pointer of its first element.

The name of an array is the pointer to the first element of the array.

If 'p' is a pointer to array 'age', means that p (or age) points to age[0].

```
int age[50];
int *p;
p = age;
```

The above code assigns 'p' the address of the first element of the array 'age'.

Address of a = address of a[0] = &a[0] = a

Now, since 'p' points to the first element of array 'age', '*p' is the value of the first element of the array.

Since *p refers to the first array element, *(p+1) and *(p+2) refers to the second and third elements respectively and so on.

So, *p is age[0], *(p+1) is age[1], *(p+2) is age[2].

Similarly, *age is age[0] (value at age), *(age+1) is age[1] (value at age+1), *(age+2) is age[2] (value at age+2) and so on.

That's all in pointer to arrays.

```
#include <stdio.h>
int main()
{
   float n[5] = { 20.4, 30.0, 5.8, 67, 15.2 }; /* declaring n as an array of 5 floats */
   float *p; /* p as a pointer to float */
   int i;
   p = n; /* p now points to array n */
   /* printing the values of elements of array */
   for (i = 0; i < 5; i++ )
   {
      printf("*(p + %d) = %f\n", i, *(p + i) );/* *(p+i) means value at (p+0),(p+1)...*/
   }
   return 0;
}
```

```
*(p + 0) = 20.400000
*(p + 1) = 30.000000
*(p + 2) = 5.800000
*(p + 3) = 67.000000
*(p + 4) = 15.200000
```

As 'p' is pointing to the first element of array,so, *p or *(p+0) represents the value at p[0] or the value at the first element of 'p'.
Similarly, *(p+1) represents value at p[1]. And *(p+3) and *(p+4) represents p[3] and p[4] respectively. So accordingly, things were printed.

```
#include <stdio.h>
int main()
{
   int n[4] = { 20, 30, 5, 67 }; /* declaring n as an array of 4 integers */
   int *p; /*a pointer*/
   int i;
   p = n; /*p is pointing to array n*/
   /* printing the address of array */
   printf("Address of array n[4] = %u\n" , p );
   /* printing the addresses of elements of array */
   for (i = 0; i < 4; i++ )
   {
      printf("Address of n[%d] = %u\n" , i, &n[i] );
   }
   return 0;
}
```

```
Address of array n[4] = 2491554384
Address of n[0] = 2491554384
Address of n[1] = 2491554388
Address of n[2] = 2491554392
Address of n[3] = 2491554396
```

Multidimensional Array

The elements of an array can be of any data type, including arrays! An array of arrays is called a **multidimensional array**.

2D Arrays

Yes, 2-dimensional arrays also exist and are generally known as matrix. These consist of rows and columns.

Before going into its application, let's first see how to declare and initialize a 2D array.

Declaration of 2D Array

Similar to one-dimensional array, we define a 2-dimensional array as below.

int a[2][4];

Here, 'a' is a 2D array of integers which consists of 2 rows and 4 columns.

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]

Assigning Values to a 2 D Array

Same as in one-dimensional array, we can assign values to the elements of a 2-dimensional array in 2 ways as well.

In the first method, just assign a value to the elements of the array. If no value is assigned to any element, then its value is assumed to be zero.

Suppose we declared a 2-dimensional array a[2][2]. Now, we need to assign values to its elements.

```
int a[2][2];
a[0][0]=1;
a[0][1]=2;
a[1][0]=3;
a[1][1]=4;
```

The second way is to declare and assign values at the same time as we did in one-dimensional array.

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

Here, value of a[0][0] is 1, a[0][1] is 2, a[0][2] is 3, a[1][0] is 4, a[1][1] is 5 and a[1][2] is 6.

Let's consider different cases of assigning values to an array at the time of declaration.

```
int a[2][2] = { 1, 2, 3, 4 }; /* valid */
int a[ ][2] = { 1, 2, 3, 4 }; /* valid */
int a[2][ ] = { 1, 2, 3, 4 }; /* invalid */
int a[ ][ ] = { 1, 2, 3, 4 }; /* invalid */
```

```
#include <stdio.h>
int main()
{
   float marks[3][2];
   int i,j;
   for( i=0; i<3; i++)
   {
      /* input of marks from the user */
      printf("Enter marks of student %d\n", (i+1) );
      for( j=0; j<2; j++)
      {
         printf("Subject %d\n", (j+1) );
         scanf("%f", &marks[i][j]);
      }
   }
   /* printing the marks of students */
   for( i=0; i<3; i++)
   {
      printf("Marks of student %d\n", (i+1) );
      for( j=0; j<2; j++)
```

```
{
        printf("Subject %d : %f\n", (j+1), marks[i][j] );
      }
   }
   return 0;
}
```

```
Enter marks of student 1
Subject 1
78
Subject 2
67
Enter marks of student 2
Subject 1
79
Subject 2
87
Enter marks of student 3
Subject 1
90
Subject 2
89
```

```
Marks of student 1
Subject 1 : 78.000000
Subject 2 : 67.000000
Marks of student 2
Subject 1 : 79.000000
Subject 2 : 87.000000
Marks of student 3
Subject 1 : 90.000000
Subject 2 : 89.000000
```

	Subject 1	Subject 2
Student 1	78	67
Student 2	79	87
Student 3	90	89

String Handling in C

The string can be defined as the one-dimensional array of characters terminated by a null ('\0'). The character array or the string is used to manipulate text such as word or sentences. Each character in the array occupies one byte of memory, and the last character must always be 0. The termination character ('\0') is important in a string since it is the only way to identify where the string ends. When we define a string as char s[10], the character s[10] is implicitly initialized with the null in the memory.

String Declaration

Method 1:

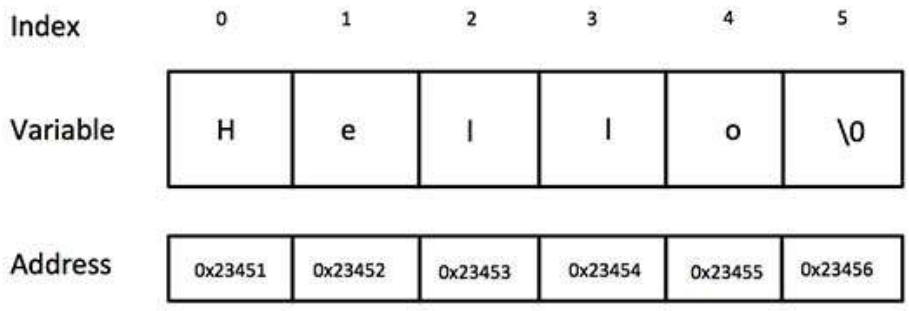
```
char address[]={'T', 'E', 'X', 'A', 'S', '\0'};
```

Method 2:

```
char address[]="TEXAS";
```

In the above declaration NULL character (\0) will automatically be inserted at the end of the string.

'\0' represents the end of the string. It is also referred as String terminator & Null Character.



Read & write Strings in C using Printf() and Scanf() functions

```
#include <stdio.h>
#include <string.h>
int main()
{
   /* String Declaration*/
   char nickname[20];

   printf("Enter your Nick name:");

   /* I am reading the input string and storing it in nickname
    * Array name alone works as a base address of array so
    * we can use nickname instead of &nickname here
    */
   scanf("%s", nickname);

   /*Displaying String*/
   printf("%s",nickname);

   return 0;
}
```

```
Enter your Nick name:Negan
Negan
```

Note: %s format specifier is used for strings input/output

Read & Write Strings in C using gets() and puts() functions

```
#include <stdio.h>
#include <string.h>
int main()
{
   /* String Declaration*/
   char nickname[20];

   /* Console display using puts */
   puts("Enter your Nick name:");
   /*Input using gets*/
   gets(nickname);
   puts(nickname);

   return 0;
}
```

```
Enter your Nick name:Negan
Negan
```

C programming language provides a set of pre-defined functions called **string handling functions** to work with string values. The string handling functions are defined in a header file called **string.h**. Whenever we want to use any string handling function we must include the header file called **string.h**.

Function	Syntax (or) Example	Description
strcpy()	strcpy(string1, string2)	Copies string2 value into string1
strncpy()	strncpy(string1, string2, 5)	Copies first 5 characters string2 into string1
strlen()	strlen(string1)	returns total number of characters in string1
strcat()	strcat(string1,string2)	Appends string2 to string1
strncat()	strncpy(string1, string2, 4)	Appends first 4 characters of string2 to string1

Function	Syntax (or) Example	Description
strcmp()	strcmp(string1, string2)	Returns 0 if string1 and string2 are the same; less than 0 if string1<string2; greater than 0 if string1>string2
strncmp()	strncmp(string1, string2, 4)	Compares first 4 characters of both string1 and string2
strcmpi()	strcmpi(string1,string2)	Compares two strings, string1 and string2 by ignoring case (upper or lower)
stricmp()	stricmp(string1, string2)	Compares two strings, string1 and string2 by ignoring case (similar to strcmpi())
strlwr()	strlwr(string1)	Converts all the characters of string1 to lower case.
strupr()	strupr(string1)	Converts all the characters of string1 to upper case.
strdup()	string1 = strdup(string2)	Duplicated value of string2 is assigned to string1

Function	Syntax (or) Example	Description
strchr()	strchr(string1, 'b')	Returns a pointer to the first occurrence of character 'b' in string1
strrchr()	'strrchr(string1, 'b')	Returns a pointer to the last occurrence of character 'b' in string1
strstr()	strstr(string1, string2)	Returns a pointer to the first occurrence of string2 in string1
strset()	strset(string1, 'B')	Sets all the characters of string1 to given character 'B'.
strnset()	strnset(string1, 'B', 5)	Sets first 5 characters of string1 to given character 'B'.
strrev()	strrev(string1)	It reverses the value of string1

The following example uses some of the above-mentioned functions −

```
#include <stdio.h>
#include <string.h>

int main () {

   char str1[12] = "Hello";
   char str2[12] = "World";
   char str3[12];
   int  len ;

   /* copy str1 into str3 */
   strcpy(str3, str1);
   printf("strcpy( str3, str1) :  %s\n", str3 );

   /* concatenates str1 and str2 */
   strcat( str1, str2);
   printf("strcat( str1, str2):   %s\n", str1 );

   /* total lenghth of str1 after concatenation */
```

```
   len = strlen(str1);
   printf("strlen(str1) :  %d\n", len );

   return 0;
}
```

```
strcpy( str3, str1) :  Hello
strcat( str1, str2):   HelloWorld
strlen(str1) :  10
```

User Defined Functions

User can define functions to do a task relevant to their programs. Such functions are called user-defined functions. The main( ) function in which we write all our program is a user-defined function. Every pro0gram execution starts at the main( ) function.
Any function (library or user-defined) has 3 things

1. Function declaration
2. Function calling
3. Function defintion

In case of library functions, the function declaration is in header files. The function is library files and the calling is in the program. But in case of user-defined functions all the 3 things are in your source program. **Function declaration:**
Syntax:
return data_type function_name(arguments list);

Function Definition:
Syntax:
return data_type function_name(argument list)
{
Body;
}

Function Calling:
Syntax:
Function_name(param_list);

Formal Arguments:
The arguments which are given at the time of function declaration or function definition are called formal arguments.
Actual Arguments:
The arguments which are given at the time of function calling are called actual arguments.

General Form of a Function

```
return data type  function-name(arglist)
        argument declaration
        {
          Local variable declaration;
          Executable statements;
          ----------------
          ----------------
          return(expression);
        }
```

All parts are not essential. Some may be absent.
For eg: The argument list and it's associated argument declaration part are optional.
We may recall that the main functions discussed so far have not included any arguments.

return: This statement is for returning a value to the calling function. This is an optional statement. It's absence indicates that no value is being return to the calling function.

Function name: A function must follow the same rules of formation as other variable names in C. Additional care must be taken to avoid duplicating library functions names or operating system commands.

Argument list : It contains valid variable name separated by commas. The list must be surrounded by parenthesis. Note that no semi colon follows the closing parenthesis. The argument variables receive values from the calling function. Thus providing a means of data communication from calling function to the called function.

If the arguments are present before beginning with the statements in the functions it is necessary to declare the type of arguments through the type declaration statements.

Note: There are two methods of declaring the parameters. The **older method** (known as **"classic"** method) declares function arguments separately after the definition of function name (known as **function header**).

The **newer method** (known as **"modern" or ANSI method**) combines the function definition and argument declaration in one line. Luckily the modern compilers support both the methods. That is, we can write a function using either of them and execute it successfully.

```
#include <stdio.h>
#include <stdlib.h>
int sumNum(int x,int y); //function declaration
int main()
{
int i,j,sum;
printf("Please enter 2 numbers for find sum\n");
scanf("%d %d",&i,&j);
sum=sumNum(i,j); //function call
printf("The result of sum is :%d",sum);
getch();
return 0;
}
int sumNum(int x, int y)//function definition
{
int result;
result=x+y;
return result; //return statements
}
```

```
Please enter 2 numbers for find sum
34
45
the result of sum is :79
```

Function Declaration

declaration of the user-defined function.

A function declaration is a frame (prototype)of function that contains the function's name, list of parameter and return type and ends with the semicolon. but it doesn't,t contain the function body

Syntax

return_Type function_Name(parameter1,parameter2,.....);

Example

Example of the function declaration

int sum_Num(int x, int y,.....);

In the above example, int sumNum(int x, int y); is a function declaration which contains the following information

Every function declaration must contain the following 3 parts and ends with the semicolon in C language

function name – name of the function is sumNum()

return type – the return type of the function is int

arguments – two arguments of type int are passed to the function

Calling a function

Syntax

function_Name(Argument list);

Example

sum_Num(argument_1, argument_2,.....);

Function Definition

The function definition is an expansion of function declaration. It contains codes in the body part of the function for execution program by the compiler – it contains the block of code for the special task

The syntax of the function definition

return_Type function_name(parameter_1, parameter_2,....){

//statements

//body of the function

}

Types of user defined functions in C

We can classify the basic function design by their return values and their parameters. Function either return a value or they don't. It may have parameters or may not. Function with doesn't return any value is called void functions. Combining return types and parameters in four basic designs:

1) Void functions with no parameter
2) Void functions with no parameters
3) Function with return value and no parameter
4) Function with return value and parameters

Void functions with no parameter

A void function may have no parameter. That function has the only task, displaying the message. That function doesn't receive anything from the caller function and returns nothing back to the caller function.

```
#include<stdio.h>
float findarea(); //function declaration
int main()
{
  findarea(); //function calling
  return 0;
}
//function to find ara of rectangle
//function definition
float findarea()
{
  float length, width, area;
  printf("Enter length and width of rectangle: ");
```

```
scanf("%f %f", &length, &width);
   area = length * width;
   printf("Area of rectangle = %.2f\n",area);
}
```

```
Enter length and width of rectangle: 10 4.23
Area of rectangle = 42.30
```

The above program calculates the area of a rectangle. One void function findarea() is defined in the program and it doesn't receive any data from the main function.

The function call still requires parentheses, however, even when no parameters are present. Without the parentheses, it is not a function call. Because a void function does not have a value, so it can be used only as a statement, It can't be used as an expression.

In this type of function, there is no data communication between calling function and called function, as well as called function and caller function.

Void functions with parameters

In this type of function, there is data communication between the calling function and called function but there is no data communication between the called function and calling function.

```
#include<stdio.h>
void findarea(float, float); //function declaration
int main()
{
   float length, width;
   printf("Enter length and width of rectangle: ");
   scanf("%f %f", &length, &width);
   findarea(length, width); //function calling
   return 0;
}
//function to find ara of rectangle
//function definition
void findarea(float l, float b)
{
   float area;
   area = l * b;
   printf("Area of rectangle = %.2f\n",area);
}
```

```
Enter length and width of rectangle: 12 4.3
Area of rectangle = 51.60
```

In this program, there is void function findarea() with parameters. The function receives two floating-point parameters. It is a void function, so it can be used only as a statement, It can't be used as an expression.

Note that the names of the variables in the main function (length and width) and the name of parameters in function (l and b) may have the same names or different names, but their data type must be the same.

User defined Function in C with return value and no parameter

```
#include<stdio.h>
float findarea(); //function declaration
int main()
{
   float result;
   result = findarea(); //function calling
   printf("Area of rectangle = %.2f\n",result);
   return 0;
}
//function to find ara of rectangle
//function definition
float findarea()
```

```
    printf("Enter length and width of rectangle: ");
    scanf("%f %f", &length, &width);
    area = length * width;
    return area;
}
```

```
Enter length and width of rectangle: 10 3.5
Area of rectangle = 35.00
```

In this program, one non-void function findarea() is defined which doesn't receive any data from the caller function and returns the area of the rectangle. The function findarea() takes input from the user, calculates the area and return the area to the caller function.

In this type of design, there is no communication between calling function and called function. But data communication exists between called function and calling function.

User-defined Function in C with return value and parameters

Unlike the three designs discussed earlier, in this type two-way data communication takes place. That is, both the called and calling functions receive and transfer data from each other. Most of the time we used this approach.

```
#include<stdio.h>
float findarea(float, float); //function declaration
int main()
{
  float length, width, result;
  printf("Enter length and width of rectangle: ");
  scanf("%f %f", &length, &width);
  result = findarea(length, width); //function calling
  printf("Area of rectangle = %.2f\n",result);
  return 0;
}
//function to find ara of rectangle
//function definition
float findarea(float l, float b)
{
  float area;
  area = l * b;
  return area; //return statement
}
```

```
Enter length and width of rectangle: 12.5 5
Area of rectangle = 62.50
```

Call by Value and Call by Reference in C

Call by value

In call by value, **original value can not be changed** or modified. In call by value, when you passed value to the function it is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only but it not change the value of variable inside the caller method such as main().

```
#include<stdio.h>
#include<conio.h>


void swap(int a, int b)
{
 int temp;
 temp=a;
 a=b;
 b=temp;
}


void main()
```

```
{
 int a=100, b=200;
 clrscr();
 swap(a, b);  // passing value to function
 printf("\nValue of a: %d",a);
 printf("\nValue of b: %d",b);
 getch();
}
```

```
Value of a: 200
Value of b: 100
```

Call by reference

In call by reference, **original value is changed** or modified because we pass reference (address). Here, address of the value is passed in the function, so actual and formal arguments shares the same address space. Hence, any value changed inside the function, is reflected inside as well as outside the function.

```
#include<stdio.h>
#include<conio.h>
void swap(int *a, int *b)
{
 int temp;
 temp=*a;
 *a=*b;
 *b=temp;
}
void main()
{
 int a=100, b=200;
 clrscr();
 swap(&a, &b);  // passing value to function
 printf("\nValue of a: %d",a);
 printf("\nValue of b: %d",b);
 getch();
}
```

```
Value of a: 200
Value of b: 100
```

Difference between call by value and call by reference.

call by value	call by reference
This method copy original value into function as a arguments.	This method copy address of arguments into function as a arguments.
Changes made to the parameter inside the function have no effect on the argument.	Changes made to the parameter affect the argument. Because address is used to access the actual argument.
Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location

Important points related to Function

- The basic purpose of the function is code reuse.
- From any function we can invoke (call) any another functions.
- Always compilation will be take place from top to bottom.
- Always execution process will starts from main() and ends with main() only.
- In implementation when we are calling a function which is define later for avoiding the compilation error we need to for forward declaration that is prototype is required.

- In function definition first line is called function declaration or function header.
- Always function declaration should be match with function declaratory.
- In implementation whenever a function does not returns any values back to the calling place then specify the return type.
- Void means nothing that is no return value.
- In implementation whenever a function returns other than void then specify the return type as return value type that is on e type of return value it is returning same type of return statement should be mentioned.
- Default return type of any function is an **int**.
- Default parameter type of any function is **void**.

Pointers in C

Pointers in C language is a variable that stores/points the address of another variable. A Pointer in C is used to allocate memory dynamically i.e. at run time. The pointer variable might be belonging to any of the data type such as int, float, char, double, short etc.

- Pointer Syntax : data_type *var_name; Example : int *p; char *p;
- Where, * is used to denote that “p” is pointer variable and not a normal variable.

KEY POINTS TO REMEMBER ABOUT POINTERS IN C:

- Normal variable stores the value whereas pointer variable stores the address of the variable.
- The content of the C pointer always be a whole number i.e. address.
- Always C pointer is initialized to null, i.e. int *p = null.
- The value of null pointer is 0.
- & symbol is used to get the address of the variable.
- symbol is used to get the value of the variable that the pointer is pointing to.
- If a pointer in C is assigned to NULL, it means it is pointing to nothing.
- Two pointers can be subtracted to know how many elements are available between these two pointers.
- But, Pointer addition, multiplication, division are not allowed.
- The size of any pointer is 2 byte (for 16 bit compiler).

```
#include <stdio.h>
int main()
{
  int *ptr, q;
  q = 50;
  /* address of q is assigned to ptr */
  ptr = &q;
  /* display q's value using ptr variable */
  printf("%d", *ptr);
  return 0;
}
```

```
50
```

Pointer Notations vs Array Notations

When we say that arrays are treated like pointers in C, we mean the following:

- The array variable holds the address of the first element in the array. It isn't a pointer but it does act like a constant pointer that cannot be changed.
- Programs often interact with arrays using pointer notation instead of array notation.

```
// initialize an array of ints
int numbers[5] = {1,2,3,4,5};

// standard array notation
int *ptr1 = numbers;
int val1 = numbers[0];

// address of array notation
int *ptr2 = &numbers[0];
int val2 = *(&numbers[0]);

// pointer + increment notation
int *ptr3 = numbers + 0;
int val3 = *(numbers + 0);

// print out the address stored in the pointers
printf("*ptr1 = %p\n", (void *)ptr1);
printf("*ptr2 = %p\n", (void *)ptr2);
```

```
printf("*ptr3 = %p\n", (void *)ptr3);

// print out the value at the pointer addresses
printf("val1 = %d\n", val1);
printf("val2 = %d\n", val1);
printf("val3 = %d\n", val1);
```

```
*ptr1 = 0x7fff6be1de60

*ptr2 = 0x7fff6be1de60

*ptr3 = 0x7fff6be1de60

val1 = 1

val2 = 1

val3 = 1
```

We declare an int array with 5 ints and assign the array numbers variable to our int pointer, ptr1. The numbers variable holds the address of the first element in the array. Assigning it to ptr1 numbers is treated as an pointer. We then get the value of the first element in the array using array notation.

In the second example we get the address of the first element in the array using array notation and then we get the value of the first element by dereferencing the address of the of the first element in the array.

In the third example we use pointer math to assign the first address of the first element in the array and we dereference the same address to get the value.

```
// initialize an array of ints
int numbers[5] = {1,2,3,4,5};
int i = 0;

// print out elements using array notation
for (i = 0; i < 5; i++ ) {
  int value = numbers[i];
  printf("numbers[%d] = %d\n", i, value);
}

// print out elements using pointer math + array indexing (yuck!)
for (i = 0; i < 5; i++ ) {
  int value = *(numbers + i);
  printf("*(numbers + %d) = %d\n", i, value);
}

// print out elements using a single pointer
int *ptr = numbers;
for (i = 0; i < 5; i++ ) {
  int value = *ptr++;
  printf("%d, *ptr++ = %d\n", i, value);
}
```

```
numbers[0] = 1
numbers[1] = 2
numbers[2] = 3
numbers[3] = 4
numbers[4] = 5
*(numbers + 0) = 1
*(numbers + 1) = 2
*(numbers + 2) = 3
*(numbers + 3) = 4
*(numbers + 4) = 5
0, *ptr++ = 1
1, *ptr++ = 2
2, *ptr++ = 3
```

Array notation is pointer arithmetic. The C standard defines that numbers[0] is just syntactic sugar for *(numbers + 0). Anytime you write array notation such as numbers[2] the compiler switches that to *(numbers + 2), where numbers is the address of the first element in the array and + 2 increments the address through pointer math.

```
// initialize an array of ints
int numbers[5] = {1,2,3,4,5};
int numbers2[5] = {6,7,8,9,0};
int *ptr = numbers2;
```

```
int *ptr = numbers2;

// this won't compile
numbers = numbers2;
numbers = &numbers2;
numbers = ptr;
```

```
incompatible types when assigning to type ‘int[5]’ from type ‘int *’

incompatible types when assigning to type ‘int[5]’ from type ‘int (*)[5]’

incompatible types when assigning to type ‘int[5]’ from type ‘int *’
```

Even though the array varible holds the address to the first element in the array, it acts like as a constant pointer in that it cannot be changed. It cannot be assigned a different array or a pointer to a different array. Think about it, if you have a variable A that points to an array and you were able to change the address of A to something else, what happens to the memory pointed to by the A array.

```
// initialize an array of ints
int numbers[5] = {1,2,3,4,5};
int numbers2[5] = {6,7,8,9,0};
int *ptr1 = numbers;
int *ptr2 = numbers2;

// this will compile
ptr1 = ptr2;
```

```
// print the addresses
printf("numbers = %p\n", (void *)numbers);
printf("numbers2 = %p\n", (void *)numbers2);
printf("ptr1 = %p\n", (void *)ptr1);
printf("ptr2 = %p\n", (void *)ptr2);
```

```
numbers = 0x7fff5ea3d230

numbers2 = 0x7fff5ea3d250

ptr1 = 0x7fff5ea3d250

ptr2 = 0x7fff5ea3d250
```

Even though we can't change the array variable directly, we can have a pointer to the array and then change that pointer. Here we create two arrays and two int pointers. We assign the numbers variable to ptr1 and numbers2 variable to ptr2. We then assign ptr2 to ptr1. Finally we print the output we can see that ptr1 and ptr2 are both pointing to the first element of the numbers2 array.

Structures

C Structure is a collection of different data types which are grouped together and each element in a C structure is called member.

- If you want to access structure members in C, structure variable should be declared.
- Many structure variables can be declared for same structure and memory will be allocated for each separately.
- It is a best practice to initialize a structure to null while declaring, if we don't assign any values to structure members.

Syntax	struct student
	{
	int a;
	char b[10];
	}

Example	a = 10; b = "Hello";

BELOW TABLE EXPLAINS FOLLOWING CONCEPTS IN C STRUCTURE.

1. How to declare a C structure?
2. How to initialize a C structure?
3. How to access the members of a C structure?

Using normal variable	Using pointer variable

Syntax:
struct tag_name
{
data type var_name1;
data type var_name2;
data type var_name3;
};
	Syntax:
	struct tag_name
	{
	data type var_name1;
	data type var_name2;
	data type var_name3;
	};

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

Declaring structure using normal variable: struct student report;	**Declaring structure using pointer variable:** struct student *report, rep;
Initializing structure using normal variable: struct student report = {100, “Mani”, 99.5};	**Initializing structure using pointer variable:** struct student rep = {100, “Mani”, 99.5}; report = &rep;
Accessing structure members	**Accessing structure members**

using normal variable:	using pointer variable:
report.mark; report.name; report.average;	report -> mark; report -> name; report -> average;

```
#include <stdio.h>
#include <string.h>

struct student
{
        int id;
        char name[20];
        float percentage;
};

int main()
{
        struct student record = {0}; //Initializing to null

        record.id=1;
        strcpy(record.name, "Raju");
        record.percentage = 86.5;

        printf(" Id is: %d \n", record.id);
        printf(" Name is: %s \n", record.name);
        printf(" Percentage is: %f \n", record.percentage);
        return 0;
}
```

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

ANOTHER WAY OF DECLARING C STRUCTURE:

```
#include <stdio.h>
#include <string.h>

struct student
{
        int id;
        char name[20];
        float percentage;
} record;

int main()
{

        record.id=1;
        strcpy(record.name, "Raju");
        record.percentage = 86.5;

        printf(" Id is: %d \n", record.id);
        printf(" Name is: %s \n", record.name);
        printf(" Percentage is: %f \n", record.percentage);
        return 0;
}
```

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

USES OF STRUCTURES IN C:

- C Structures can be used to store huge data. Structures act as a database.
- C Structures can be used to send data to the printer.
- C Structures can interact with keyboard and mouse to store the data.
- C Structures can be used in drawing and floppy formatting.
- C Structures can be used to clear output screen contents.
- C Structures can be used to check computer's memory size etc.

Unions, Typedef and Enumeration

Unions

Union is an user defined datatype in C programming language. It is a collection of variables of different datatypes in the same memory location. We can define a union with many members, but at a given point of time only one member can contain a value. Unions can be very handy when you need to talk to peripherals through some memory mapped registers.

Difference between structure and union

The main difference between structure and a union is that

- Structs allocate enough space to store all of the fields in the struct. The first one is stored at the beginning of the struct, the second is stored after that, and so on.
- Unions only allocate enough space to store the largest field listed, and all fields are stored at the same space .

Syntax for Declaring a C union

```
union union_name
{ datatype field_name;
  datatype field_name; // more variables
};
```

To access the fields of a union, use dot(.) operator i.e., the variable name followed by dot operator followed by field name.

```
#include <stdio.h>
union item
{
   int a;
   float b;
   char ch;
};
int main( )
{
   union item it;
   it.a = 12;
   it.b = 20.2;
   it.ch = 'z';

   printf("%d\n", it.a);
   printf("%f\n", it.b);
   printf("%c\n", it.ch);
   return 0;
}
```

```
-26426
20.1999
z
```

Typedef

In C programming language, **typedef** is a keyword used to create alias name for the existing datatypes. Using **typedef** keyword we can create a temporary name to the system defined datatypes like int, float, char and double. we use that temporary name to create a variable. The general syntax of typedef is as follows : -

typedef <existing-datatype> <alias-name>

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;


type_name t1, t2;
```

```
#include<stdio.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);
}
```

Enumeration

Enumeration (enum) is a user-defined datatype (same as structure). It consists of various elements of that type. There is no such specific use of enum, we use it just to make our codes neat and more readable. We can write C programs without using enumerations also.

An enum is defined in the same way as structure with the keyword struct replaced by the keyword enum and the elements separated by 'comma' as follows.

```
enum enum_name
{
  element1,
  element2,
  element3,
  element4,
};
```

We also declare an enum variable in the same way as that of structures. We create an enum variable as follows.

```
enum season

{
  Summer,
  Spring,
  Winter,
  Autumn
};
```

```
main()
{
enum season s;
}
```

So, here 's' is the variable of the enum named season. This variable will represent a season. We can also declare an enum variable as follows.

```
enum season{
  Summer,
  Spring,
  Winter,
  Autumn
}s;
```

```
#include <stdio.h>
enum season{ Summer, Spring, Winter, Autumn};
int main()
{
    enum season s;
    s = Spring;
    printf("%d\n",s);
    return 0;
}
```

```
1
```

Here, first we defined an enum named 'season' and declared its variable 's' in the main function as we have seen before. The values of Summer, Spring, Winter and Autumn are 0, 1, 2 and 3 respectively. So, by writing s = Spring, we assigned a value '1' to the variable 's' since the value of 'Spring' is 1.

```
#include <stdio.h>
enum days{ sun, mon, tue = 5, wed, thurs, fri, sat};
int main()
{
   enum days day;
   day = thurs;
   printf("%d\n",day);
   return 0;
}
```

```
7
```

The default value of 'sun' will be 0, 'mon' will be 1, 'tue' will be 2 and so on. In the above example, we defined the value of tue as 5. So the values of 'wed', 'thurs', 'fri' and 'sat' will become 6, 7, 8 and 9 respectively. There will be no effect on the values of sun and mon which will remain 0 and 1 respectively. Thus the value of thurs i.e. 7 will get printed.

```
#include <stdio.h>
enum days{ sun, mon, tue, wed, thurs, fri, sat};
int main()
{
    enum days day;
    day = thurs;
    printf("%d\n",day+2);
    return 0;
}
```

```
6
```

In this example, the value of 'thurs' i.e. 4 is assigned to the variable day. Since we are printing 'day+2' i.e. 6 (=4+2), so the output will be 6.

Pre Processor Commands

The C pre-processor is a macro pre-processor (allows you to define macros) that transforms your program before it is compiled. These transformations can be the inclusion of header file, macro expansions etc.

C Macros

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

Object-like Macros

- Function-like Macros
- Object-like Macros

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

#define PI 3.14

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

Function-like Macros

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

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

Here, MIN is the macro name.

C Predefined Macros

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

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

```
#include<stdio.h>
 int main(){
   printf("File :%s\n", __FILE__ );
   printf("Date :%s\n", __DATE__ );
   printf("Time :%s\n", __TIME__ );
   printf("Line :%d\n", __LINE__ );
   printf("STDC :%d\n", __STDC__ );
   return 0;
 }
```

```
File :simple.c
Date :Dec 6 2015
Time :12:28:46
Line :6
STDC :1
```

C #include

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

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

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

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

```
#include<stdio.h>
 int main(){
   printf("Hello C");
   return 0;
 }
```

```
Hello C
```

C# define

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

Syntax:

#define token value

```
#include <stdio.h>
#define PI 3.14
main() {
   printf("%f",PI);
}
```

```
3.140000
```

C #undef

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

Syntax:

#undef token

```
#include <stdio.h>
#define PI 3.14
#undef PI
main() {
   printf("%f",PI);
}
```

Compile Time Error: 'PI' undeclared

```
#include <stdio.h>
#define number 15
int square=number*number;
#undef number
main() {
   printf("%d",square);
}
```

225

File Handling

A File can be used to store a large volume of persistent data. Like many other languages 'C' provides following file management functions,

1. Creation of a file
2. Opening a file
3. Reading a file
4. Writing to a file
5. Closing a file

Function	Purpose
fopen ()	Creating a file or opening an existing file
fclose ()	Closing a file
fprintf ()	Writing a block of data to a file
fscanf ()	Reading a block data from a file
getc ()	Reads a single character from a file
putc ()	Writes a single character to a file
getw ()	Reads an integer from a file
putw ()	Writing an integer to a file
fseek ()	Sets the position of a file pointer to a specified location
ftell ()	Returns the current position of a file pointer
rewind ()	Sets the file pointer at the beginning of a file

Create a File

Whenever you want to work with a file, the first step is to create a file. A file is nothing but space in a memory where data is stored.To create a file in a 'C' program following syntax is used,

```
FILE *fp;

fp = fopen ("file_name", "mode");
```

In the above syntax, the file is a data structure which is defined in the standard library. fopen is a standard function which is used to open a file.

- If the file is not present on the system, then it is created and then opened.
- If a file is already present on the system, then it is directly opened using this function.

fp is a file pointer which points to the type file.

Whenever you open or create a file, you have to specify what you are going to do with the file. A file in 'C' programming can be created or opened for reading/writing purposes. A mode is used to specify whether you want to open a file for any of the below-given purposes. Following are the different types of modes in 'C' programming which can be used while working with a file.

In the given syntax, the filename and the mode are specified as strings hence they must always be enclosed within double quotes.

File Mode	Description
r	Open a file for reading. If a file is in reading mode, then no data is deleted if a file is already present on a system.
w	Open a file for writing. If a file is in writing mode, then a new file is created if a file doesn't exist at all. If a file is already present on a system, then all the data inside the file is truncated, and it is opened for writing purposes.
a	Open a file in append mode. If a file is in append mode, then the file is opened. The content within the file doesn't change.
r+	open for reading and writing from beginning
w+	open for reading and writing, overwriting a file
a+	open for reading and writing, appending to file

```
#include <stdio.h>
int main() {
FILE *fp;
fp  = fopen ("data.txt", "w");
}
```

```
#include <stdio.h>
int main() {
FILE *fp;
fp  = fopen ("D://data.txt", "w");
}
```

Close a file

One should always close a file whenever the operations on file are over. It means the contents and links to the file are terminated. This prevents accidental damage to the file.'C' provides the fclose function to perform file closing operation. The syntax of fclose is as follows,

fclose (file_pointer);

Example:

```
FILE *fp;
fp  = fopen ("data.txt", "r");
fclose (fp);
```

The fclose function takes a file pointer as an argument. The file associated with the file pointer is then closed with the help of fclose function. It returns 0 if close was successful and EOF (end of file) if there is an error has occurred while file closing.

After closing the file, the same file pointer can also be used with other files.

Writing to a File

In C, when you write to a file, newline characters '\n' must be explicitly added.

The stdio library offers the necessary functions to write to a file:

- **fputc(char, file_pointer)**: It writes a character to the file pointed to by file_pointer.
- **fputs(str, file_pointer)**: It writes a string to the file pointed to by file_pointer.
- **fprintf(file_pointer, str, variable_lists)**: It prints a string to the file pointed to by file_pointer. The string can optionally include format specifiers and a list of variables variable_lists.

fputc

```
#include <stdio.h>
int main() {
        int i;
        FILE * fptr;
        char fn[50];
        char str[] = "TechnoXamm Rocks\n";
        fptr = fopen("fputc_test.txt", "w"); // "w" defines "writing mode"
        for (i = 0; str[i] != '\n'; i++) {
            /* write to file using fputc() function */
            fputc(str[i], fptr);
        }
        fclose(fptr);
        return 0;
    }
```

fputs

```
#include <stdio.h>
int main() {
        FILE * fp;
        fp = fopen("fputs_test.txt", "w+");
        fputs("This is C Tutorial on fputs,", fp);
        fputs("We don't need to use for loop\n", fp);
        fputs("Easier than fputc function\n", fp);
        fclose(fp);
        return (0);
    }
```

fprint

```
#include <stdio.h>
    int main() {
        FILE *fptr;
        fptr = fopen("fprintf_test.txt", "w"); // "w" defines "writing
mode"
        /* write to file */
        fprintf(fptr, "Learning C with TechnoXamm\n");
        fclose(fptr);
        return 0;
    }
```

Reading data from a File

There are three different functions dedicated to reading data from a file

- **fgetc(file_pointer):** It returns the next character from the file pointed to by the file pointer. When the end of the file has been reached, the EOF is sent back.
- **fgets(buffer, n, file_pointer):** It reads n-1 characters from the file and stores the string in a buffer in which the NULL character '\0' is appended as the last character.
- **fscanf(file_pointer, conversion_specifiers, variable_adresses)**: It is used to parse and analyze data. It reads characters from the file and assigns the input to a list of variable pointers variable_adresses using conversion specifiers. Keep in mind that as with scanf, fscanf stops reading a string when space or newline is encountered.

```
#include <stdio.h>
int main() {
        FILE * file_pointer;
        char buffer[30], c;


        file_pointer = fopen("fprintf_test.txt", "r");
        printf("----read a line----\n");
        fgets(buffer, 50, file_pointer);
        printf("%s\n", buffer);
```

```
printf("----read and parse data----\n");
        file_pointer = fopen("fprintf_test.txt", "r"); //reset the pointer
        char str1[10], str2[2], str3[20], str4[2];
        fscanf(file_pointer, "%s %s %s %s", str1, str2, str3, str4);
        printf("Read String1 |%s|\n", str1);
        printf("Read String2 |%s|\n", str2);
        printf("Read String3 |%s|\n", str3);
        printf("Read String4 |%s|\n", str4);

        printf("----read the entire file----\n");
```

```
----read a line----
Learning C with TechnoXamm

----read and parse data----
Read String1 |Learning|
Read String2 |C|
Read String3 |with|
Read String4 |TechnoXamm|
----read the entire file----
```

Interactive File Read and Write with getc and putc

These are the simplest file operations. Getc stands for get character, and putc stands for put character. These two functions are used to handle only a single character at a time.

```
#include <stdio.h>
int main() {
	FILE * fp;
	char c;
	printf("File Handling\n");
	//open a file
	fp = fopen("demo.txt", "w");
	//writing operation
	while ((c = getchar()) != EOF) {
	    putc(c, fp);
	}
	//close file
	fclose(fp);
	printf("Data Entered:\n");
	//reading
	fp = fopen("demo.txt", "r");
	while ((c = getc(fp)) != EOF) {
```

```
        printf("%c", c);
    }
    fclose(fp);
    return 0;
}
```

- In the above program we have created and opened a file called demo in a write mode.
- After a write operation is performed, then the file is closed using the fclose function.
- We have again opened a file which now contains data in a reading mode. A while loop will execute until the eof is found. Once the end of file is found the operation will be terminated and data will be displayed using printf function.
- After performing a reading operation file is again closed using the fclose function.

Command Line Arguments

Command line arguments are nothing but simply arguments that are specified after the name of the program in the system's command line, and these argument values are passed on to your program during program execution.

Components of Line Arguments

In order to implement command line arguments, generally, 2 parameters are passed into the main function:

1. Number of command line arguments
2. The list of command line arguments

The basic syntax is:

```
int main( int argc, char *argv[] )
{
.
.
// BODY OF THE MAIN FUNCTION
.
.
}
```

```
int main( int argc, char **argv[] )
{
.
.
// BODY OF THE MAIN FUNCTION
.
.
}
```

- **argc:** It refers to “argument count”. It is the first parameter that we use to store the number of command line arguments. It is important to note that the value of argc should be greater than or equal to 0.
- **agrv:** It refers to “argument vector”. It is basically an array of character pointer which we use to list all the command line arguments.

```
// The program name is cl.c
#include<stdio.h>
int main(int argc, char** argv)
{

printf("Welcome to DataFlair tutorials!\n\n");

int i;
printf("The number of arguments are: %d\n",argc);
printf("The arguments are:");

for ( i = 0; i < argc; i++)
{
printf("%s\n", argv[i]);
}
return 0;
}
```

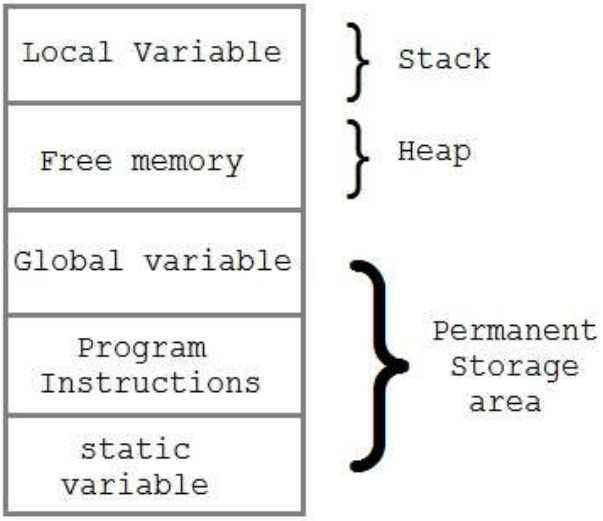
Dynamic Memory Allocation

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** header file.

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 returns a void pointer to the memory
free	releases previously allocated memory
realloc	modify the size of previously allocated space

Global variables, static variables and program instructions get their memory in **permanent** storage area whereas **local** variables are stored in a memory 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 the 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 allocate enough space as specified, it returns a NULL pointer.

Syntax:

void* malloc(byte-size)

```
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 allocate enough space as specified, it returns a NULL pointer.

Syntax:

void *calloc(number of items, element-size)

```
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 dynamically to a variable.

Syntax:

void* realloc(pointer, new-size)

```
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);

Linked List

Linked lists are the best and simplest example of a dynamic data structure that uses pointers for its implementation. However, understanding pointers is crucial to understanding how linked lists work, so if you've skipped the pointers tutorial, you should go back and redo it. You must also be familiar with dynamic memory allocation and structures.

Essentially, linked lists function as an array that can grow and shrink as needed, from any point in the array.

Linked lists have a few advantages over arrays:

1. Items can be added or removed from the middle of the list
2. There is no need to define an initial size

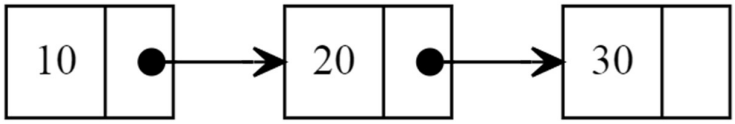
However, linked lists also have a few disadvantages:

1. There is no "random" access - it is impossible to reach the nth item in the array without first iterating over all items up until that item. This means we have to start from the beginning of the list and count how many times we advance in the list until we get to the desired item.
2. Dynamic memory allocation and pointers are required, which complicates the code and increases the risk of memory leaks and segment faults.
3. Linked lists have a much larger overhead over arrays, since linked list items are dynamically allocated (which is less efficient in memory usage) and each item in the list also must store an additional pointer.

Linked List

A linked list is a set of dynamically allocated nodes, arranged in such a way that each node contains one value and one pointer. The pointer always points to the next member of the list. If the pointer is NULL, then it is the last node in the list.

A linked list is held using a local pointer variable which points to the first item of the list. If that pointer is also NULL, then the list is considered to be empty.



Linked List

Let's define a linked list node:

```
typedef struct node {
    int val;
    struct node * next;
} node_t
```

Notice that we are defining the struct in a recursive manner, which is possible in C. Let's name our node type node_t.

Now we can use the nodes. Let's create a local variable which points to the first item of the list (called head).

```
node_t * head = NULL;
head = (node_t *) malloc(sizeof(node_t));
if (head == NULL) {
    return 1;
}
```

We've just created the first variable in the list. We must set the value, and the next item to be empty, if we want to finish populating the list. Notice that we should always check if malloc returned a NULL value or not.

To add a variable to the end of the list, we can just continue advancing to the next pointer:

```
node_t * head = NULL;
head = (node_t *) malloc(sizeof(node_t));
head->val = 1;
head->next = (node_t *) malloc(sizeof(node_t));
head->next->val = 2;
head->next->next = NULL;
```

Iterating over a list

Let's build a function that prints out all the items of a list. To do this, we need to use a current pointer that will keep track of the node we are currently printing. After printing the value of the node, we set the current pointer to the next node, and print again, until we've reached the end of the list (the next node is NULL).

```
void print_list(node_t * head) {
    node_t * current = head;

    while (current != NULL) {
        printf("%d\n", current->val);
        current = current->next;
    }
}
```

Adding an item to the end of the list

To iterate over all the members of the linked list, we use a pointer called current. We set it to start from the head and then in each step we advance the pointer to the next item in the list until we reach the last item.

```
void push(node_t * head, int val) {
    node_t * current = head;
    while (current->next != NULL) {
        current = current->next;
```

Adding an item to the beginning of the list (pushing to the list)
To add to the beginning of the list, we will need to do the following:

1. Create a new item and set its value
2. Link the new item to point to the head of the list
3. Set the head of the list to be our new item

This will effectively create a new head to the list with a new value, and keep the rest of the list linked to it.

Since we use a function to do this operation, we want to be able to modify the head variable. To do this, we must pass a pointer to the pointer variable (a double pointer) so we will be able to modify the pointer itself.

```
void push(node_t ** head, int val) {
    node_t * new_node;
    new_node = (node_t *) malloc(sizeof(node_t));

    new_node->val = val;
    new_node->next = *head;
    *head = new_node;
}
```

Removing the first item (popping from the list)

To pop a variable, we will need to reverse this action:

1. Take the next item that the head points to and save it
2. Free the head item
3. Set the head to be the next item that we've stored on the side

```
int pop(node_t ** head) {
    int retval = -1;
    node_t * next_node = NULL;

    if (*head == NULL) {
        return -1;
    }
    next_node = (*head)->next;
    retval = (*head)->val;
    free(*head);
    *head = next_node;
    return retval;
}
```

Removing the last item of the list

Removing the last item from a list is very similar to adding it to the end of the list, but with one big exception - since we have to change one

item before the last item, we actually have to look two items ahead and see if the next item is the last one in the list:

```
int remove_last(node_t * head) {
    int retval = 0;
    /* if there is only one item in the list, remove it */
    if (head->next == NULL) {
        retval = head->val;
        free(head);
        return retval;
    }
    /* get to the second to last node in the list */
    node_t * current = head;
    while (current->next->next != NULL) {
        current = current->next;
    }
    /* now current points to the second to last item of the list, so
let's remove current->next */
    retval = current->next->val;
    free(current->next);
    current->next = NULL;
    return retval;
}
```

Removing a specific item

To remove a specific item from the list, either by its index from the beginning of the list or by its value, we will need to go over all the items, continuously looking ahead to find out if we've reached the node before the item we wish to remove. This is because we need to change the location to where the previous node points to as well.

Here is the algorithm:

1. Iterate to the node before the node we wish to delete
2. Save the node we wish to delete in a temporary pointer
3. Set the previous node's next pointer to point to the node after the node we wish to delete
4. Delete the node using the temporary pointer

```
int remove_by_index(node_t ** head, int n) {
    int i = 0;
    int retval = -1;
    node_t * current = *head;
    node_t * temp_node = NULL;
    if (n == 0) {
        return pop(head);
    }
    for (i = 0; i < n-1; i++) {
        if (current->next == NULL) {
```

```
return -1;
        }
        current = current->next;
    }
    temp_node = current->next;
    retval = temp_node->val;
    current->next = temp_node->next;
    free(temp_node);

    return retval;
}
```