UNIT-1

C++ OOPs Concepts

The major purpose of C++ programming is to introduce the concept of object orientation to the C programming language.

Object Oriented Programming is a paradigm that provides many concepts such as **inheritance, data binding, polymorphism etc.**

The programming paradigm where everything is represented as an object is known as truly object-oriented programming language. **Smalltalk** is considered as the first truly object-oriented programming language.

OOPs (Object Oriented Programming System)

Object means a real word entity such as pen, chair, table etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

Class

Collection of objects is called class. It is a logical entity.

Inheritance

When one object acquires all the properties and behaviours of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

When **one task is performed by different ways** i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.

In C++, we use Function overloading and Function overriding to achieve polymorphism.

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing.

In C++, we use abstract class and interface to achieve abstraction.

Encapsulation

Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.

Advantage of OOPs over Procedure-oriented programming language

1. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.
2. OOPs provide data hiding whereas in Procedure-oriented programming language a global data can be accessed from anywhere.
3. OOPs provide ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.
4. Both Procedural Oriented Programming (POP) and Object Oriented Programming (OOP) are the high level languages in programming world and are widely used in development of applications. On the basis of nature of developing the code both languages have different approaches on basis of which both are differentiate from each other.
5. Following are the important differences between Procedural Oriented Programming (POP) and Object Oriented Programming (OOP)

Sr. No.	Key	Object Oriented Programming (OOP)	Procedural Oriented Programming (POP)
1	Definition	Object-oriented Programming is a programming language that uses classes and objects to create models based on the real world environment. In OOPs it makes it easy to maintain and modify existing code as new objects are created inheriting characteristics from existing ones.	On other hand Procedural Oriented Programming is a programming language that follows a step-by-step approach to break down a task into a collection of variables and routines (or subroutines) through a sequence of instructions. Each step is carried out in order in a systematic manner so that a computer can understand what to do.
2	Approach	In OOPs concept of objects and classes is introduced and hence the program is divided into small chunks called objects which are instances of classes.	On other hand in case of POP the main program is divided into small parts based on the functions and is treated as separate program for individual smaller program.
3	Access	In OOPs access modifiers are introduced namely as Private,	On other hand no such modifiers are

Sr. No.	Key	Object Oriented Programming (OOP)	Procedural Oriented Programming (POP)
	modifiers	public, and Protected.	introduced in POP.
4	Security	Due to abstraction in OOPs data hiding is possible and hence it is more secure than POP.	On other hand POP is less secure as compare to OOPs.
5	Complexity	OOPs due to modularity in its programs is less complex and hence new data objects can be created easily from existing objects making object-oriented programs easy to modify	On other hand there is no simple process to add data in POP at least not without revising the whole program.

C++ history

History of C++ language is interesting to know. Here we are going to discuss brief history of C++ language.

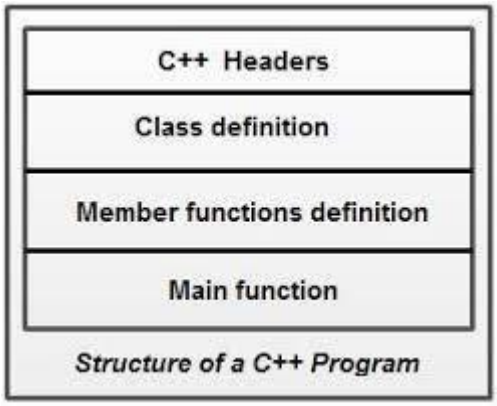
C++ programming language was developed in 1980 by Bjarne Stroustrup at bell laboratories of AT&T (American Telephone & Telegraph), located in U.S.A.

Bjarne Stroustrup is known as the **founder of C++ language.**

It was develop for adding a feature of **OOP (Object Oriented Programming)** in C without significantly changing the C component.

C++ programming is "relative" (called a superset) of C, it means any valid C program is also a valid C++ program.

Structure of C++ program:



Structure of a C++ Program

C++ Features

C++ is object oriented programming language. It provides a lot of **features** that are given below.

1. Simple
2. Machine Independent or Portable
3. Mid-level programming language
4. Structured programming language
5. Rich Library
6. Memory Management
7. Fast Speed
8. Pointers
9. Recursion
10. Extensible
11. Object Oriented
12. Compiler based

1) Simple

C++ is a simple language in the sense that it provides structured approach (to break the problem into parts), rich set of library functions, data types etc.

2) Machine Independent or Portable

Unlike assembly language, c programs can be executed in many machines with little bit or no change. But it is not platform-independent.

3) Mid-level programming language

C++ is also used to do low level programming. It is used to develop system applications such as kernel, driver etc. It also supports the feature of high level language. That is why it is known as mid-level language.

4) Structured programming language

C++ is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify.

5) Rich Library

C++ provides a lot of inbuilt functions that makes the development fast.

6) Memory Management

It supports the feature of dynamic memory allocation. In C++ language, we can free the allocated memory at any time by calling the free() function.

7) Speed

The compilation and execution time of C++ language is fast.

8) Pointer

C++ provides the feature of pointers. We can directly interact with the memory by using the pointers. We can use pointers for memory, structures, functions, array etc.

9) Recursion

In C++, we can call the function within the function. It provides code reusability for every function.

10) Extensible

C++ language is extensible because it can easily adopt new features.

C++ Basic Input/Output

C++ I/O operation is using the stream concept. Stream is the sequence of bytes or flow of data. It makes the performance fast.

If bytes flow from main memory to device like printer, display screen, or a network connection, etc, this is called as **output operation.**

If bytes flow from device like printer, display screen, or a network connection, etc to main memory, this is called as **input operation.**

I/O Library Header Files

Let us see the common header files used in C++ programming are:

Header File	Function and Description
<iostream.h>	It is used to define the **cout, cin and cerr** objects, which correspond to standard output stream, standard input stream and standard error stream, respectively.
<iomanip>	It is used to declare services useful for performing formatted I/O, such as **setprecision and setw.**
<fstream>	It is used to declare services for user-controlled file processing.

Standard output stream (cout)

The **cout** is a predefined object of **ostream** class. It is connected with the standard output device, which is usually a display screen. The cout is used in conjunction with stream insertion operator (<<) to display the output on a console

Standard input stream (cin)

The **cin** is a predefined object of **istream** class. It is connected with the standard input device, which is usually a keyboard. The cin is used in conjunction with stream extraction operator (>>) to read the input from a console.

Let's see the simple example of standard input stream (cin):

Standard end line (endl)

The **endl** is a predefined object of **ostream** class. It is used to insert a new line characters and flushes the stream.

Let's see the simple example of standard end line (endl):

```
#include <iostream.h.h>

int main( ) {
cout << "C++ Tutorial";
cout << " Javatpoint"<<endl;
cout << "End of line"<<endl;
}
```

Output:

```
C++ Tutorial Javatpoint
End of line
```

C++ Data Types

A data type specifies the type of data that a variable can store such as integer, floating, character etc.

There are 4 types of data types in C++ language.

Types	Data Types
Basic Data Type	int, char, float, double, etc
Derived Data Type	array, pointer, etc
Enumeration Data Type	enum
User Defined Data Type	structure

Basic Data Types

The basic data types are integer-based and floating-point based. C++ language supports both signed and unsigned literals.

The memory size of basic data types may change according to 32 or 64 bit operating system.

Let's see the basic data types. It size is given according to 32 bit OS.

Data Types	Memory Size	Range
char	1 byte	-128 to 127
signed char	1 byte	-128 to 127
unsigned char	1 byte	0 to 127
short	2 byte	-32,768 to 32,767
signed short	2 byte	-32,768 to 32,767
unsigned short	2 byte	0 to 32,767
int	2 byte	-32,768 to 32,767
signed int	2 byte	-32,768 to 32,767
unsigned int	2 byte	0 to 32,767
short int	2 byte	-32,768 to 32,767
signed short int	2 byte	-32,768 to 32,767
unsigned short int	2 byte	0 to 32,767
long int	4 byte	-2,147,483,648 to 2,147,483,647
signed long int	4 byte	-2,147,483,648 to 2,147,483,647
unsigned long int	4 byte	0 to 18,446,744,073,709,551,615
float	4 byte	
double	8 byte	
long double	10 byte	

C++ Variable

A variable is a name of memory location. It is used to store data. Its value can be changed and it can be reused many times.

It is a way to represent memory location through symbol so that it can be easily identified.

Let's see the syntax to declare a variable:

```
type variable_list;
```

The example of declaring variable is given below:

```
int x;
float y;
char z;
```

Here, x, y, z are variables and int, float, char are data types.

We can also provide values while declaring the variables as given below:

```
int x=5,b=10;  //declaring 2 variable of integer type
float f=30.8;
char c='A';
```

Rules for defining variables

A variable can have alphabets, digits and underscore.

A variable name can start with alphabet and underscore only. It can't start with digit.

No white space is allowed within variable name.

A variable name must not be any reserved word or keyword e.g. char, float etc.

Valid variable names:

```
int a;
int _ab;
int a30;
```

Invalid variable names:

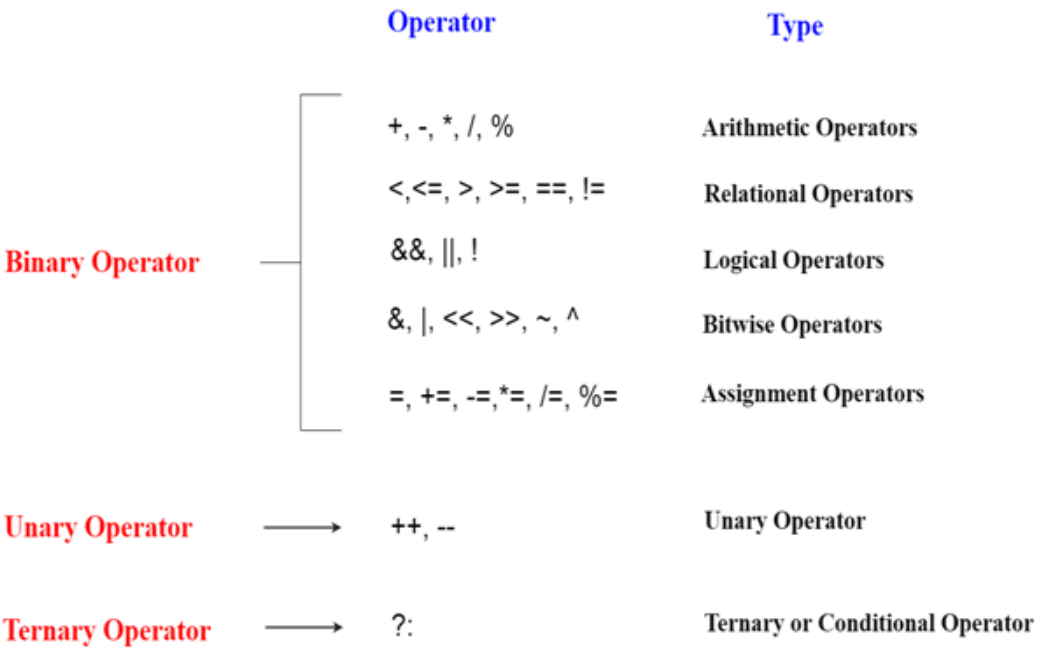
```
int 4;
int x y;
int double;
```

C++ Operators

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
- Assignment Operator
- Unary operator
- Ternary or Conditional Operator
- Misc Operator



Precedence of Operators in C++

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

Let's understand the precedence by the example given below:

1. **int** data=5+10*10;

The "data" variable will contain 105 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	+ -	Right to left
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=/td>	Right to left
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Right to left
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

C++ Identifiers

C++ identifiers in a program are used to refer to the name of the variables, functions, arrays, or other user-defined data types created by the programmer. They are the basic requirement of any language. Every language has its own rules for naming the identifiers.

In short, we can say that the C++ identifiers represent the essential elements in a program which are given below:

- **Constants**
- **Variables**
- **Functions**
- **Labels**
- **Defined data types**

C++ Control Statement

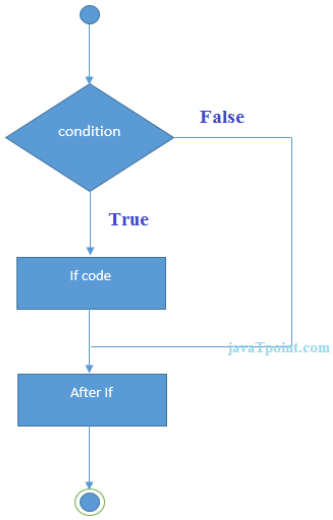
In C++ programming, if statement is used to test the condition. There are various types of if statements in C++.

- if statement
- if-else statement
- nested if statement
- if-else-if ladder

C++ IF Statement

The C++ if statement tests the condition. It is executed if condition is true.

```
if(condition){
//code to be executed
}
```



C++ If Example

```
#include <iostream.h>
  void main () {
   int num = 10;
          if (num % 2 == 0)
          {
             cout<<"It is even number";
          }
   return 0;
}
```

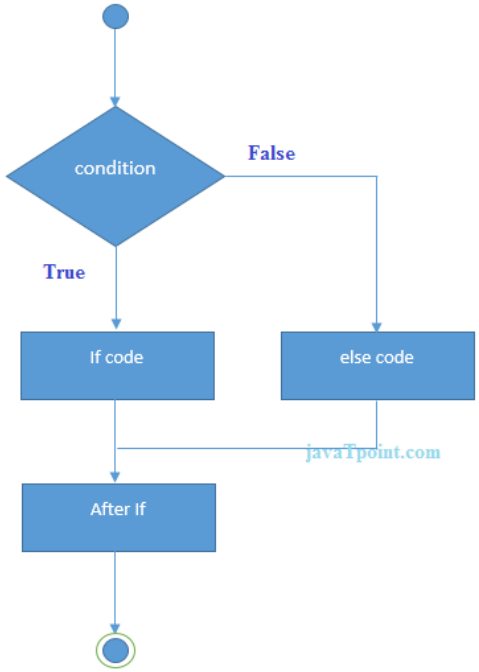
Output:

```
It is even number
```

C++ IF-else Statement

The C++ if-else statement also tests the condition. It executes if block if condition is true otherwise else block is executed.

```
if(condition){
//code if condition is true
}else{
//code if condition is false
}
```



C++ If-else Example

```
#include <iostream.h>
 void main () {
  int num = 11;
        if (num % 2 == 0)
        {
           cout<<"It is even number";
        }
        else
        {
           cout<<"It is odd number";
        }
  return 0;
}
```

Output:

```
It is odd number
```

C++ If-else Example: with input from user

```
#include <iostream.h>
void main () {
   int num;
   cout<<"Enter a Number: ";
   cin>>num;
         if (num % 2 == 0)
         {
            cout<<"It is even number"<<endl;
         }
         else
         {
            cout<<"It is odd number"<<endl;
         }
  return 0;
}
```

Output:

```
Enter a number:11
It is odd number
```

Output:

```
Enter a number:12
It is even number
```

C++ IF-else-if ladder Statement

The C++ if-else-if ladder statement executes one condition from multiple statements.

```
if(condition1){
//code to be executed if condition1 is true
}else if(condition2){
//code to be executed if condition2 is true
}
else if(condition3){
//code to be executed if condition3 is true
}
...
else{
//code to be executed if all the conditions are false
}
```

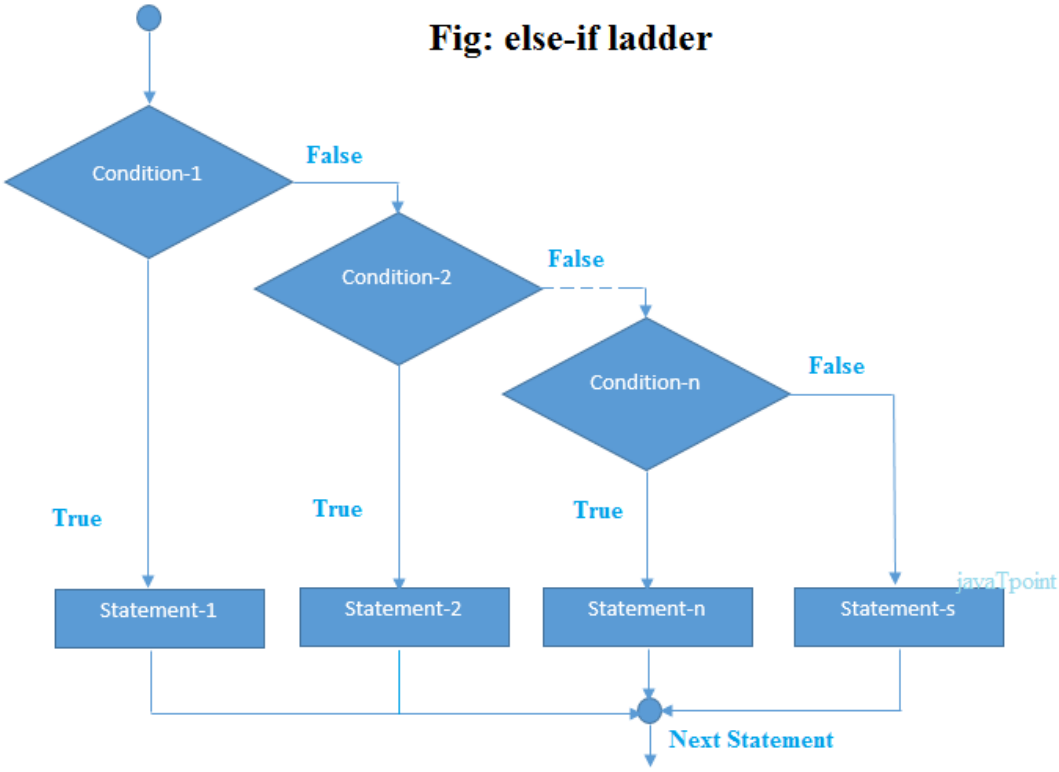


Fig: else-if ladder

C++ If else-if Example

```
#include <iostream.h>
 void main () {
                int num;
cout<<"Enter a number to check grade:";
cin>>num;
   if (num <0 || num >100)
   {
      cout<<"wrong number";
   }
   else if(num >= 0 && num < 50){
      cout<<"Fail";
   }
   else if (num >= 50 && num < 60)
   {
      cout<<"D Grade";
   }
   else if (num >= 60 && num < 70)
   {
      cout<<"C Grade";
   }
   else if (num >= 70 && num < 80)
   {
      cout<<"B Grade";
```

```
    }
    else if (num >= 80 && num < 90)
    {
       cout<<"A Grade";
    }
    else if (num >= 90 && num <= 100)
    {
       cout<<"A+ Grade";
    }
}
```

Output:

```
Enter a number to check grade:66
C Grade
```

Output:

```
Enter a number to check grade:-2
wrong number
```

C++ switch

The C++ switch statement executes one statement from multiple conditions. It is like if-else-if ladder statement in C++.

```
switch(expression){
case value1:
 //code to be executed;
 break;
case value2:
 //code to be executed;
 break;
.....

default:
 //code to be executed if all cases are not matched;
 break;
}
```

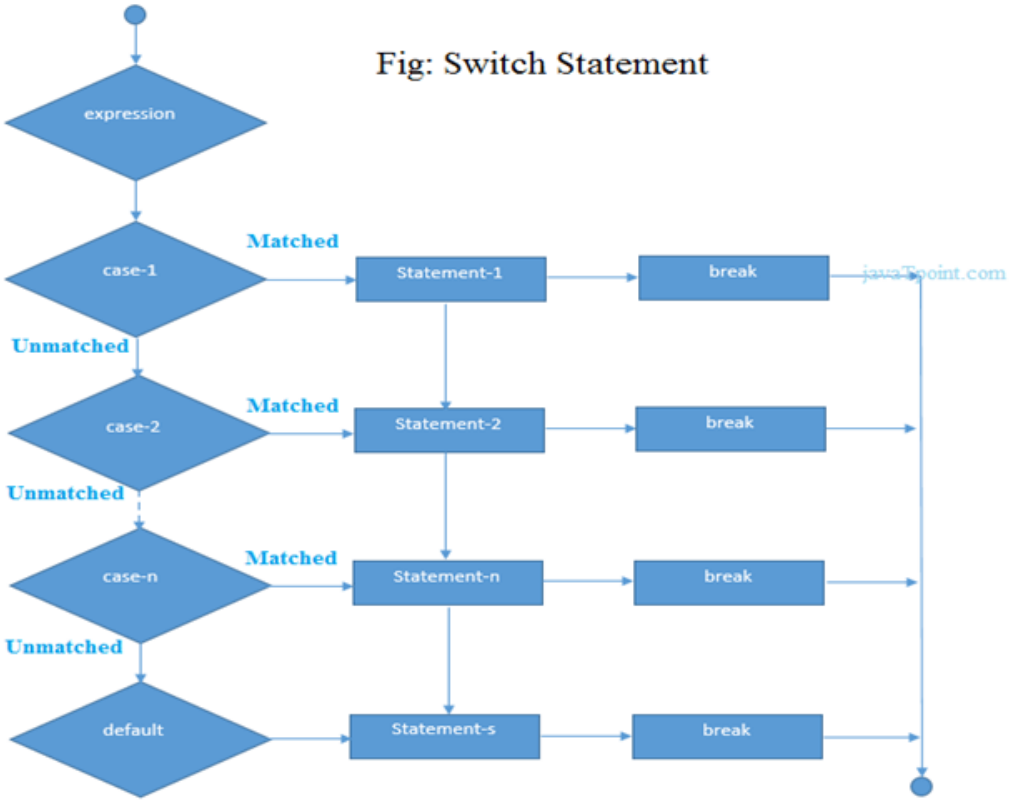


Fig: Switch Statement

C++ Switch Example

```
#include <iostream.h>
 void main () {
    int num;
    cout<<"Enter a number to check grade:";
    cin>>num;
       switch (num)
      {
         case 10: cout<<"It is 10"; break;
         case 20: cout<<"It is 20"; break;
         case 30: cout<<"It is 30"; break;
         default: cout<<"Not 10, 20 or 30"; break;
      }
  }
```

Output:

```
Enter a number:
10
It is 10
```

Output:

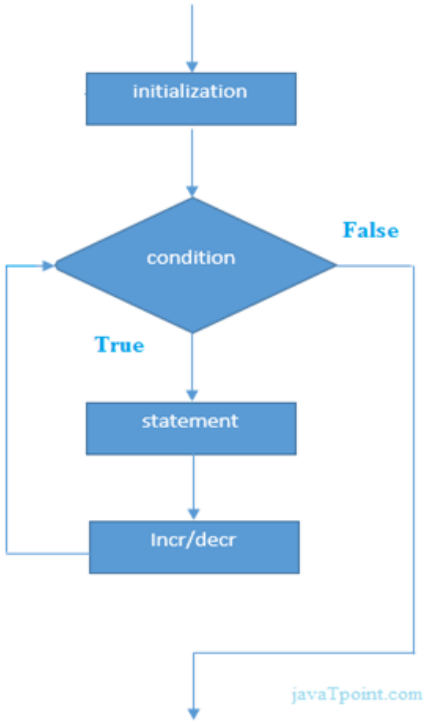
```
Enter a number:
55
Not 10, 20 or 30
```

C++ For Loop

The C++ for loop is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop than while or do-while loops.

The C++ for loop is same as C/C#. We can initialize variable, check condition and increment/decrement value.

```
for(initialization; condition; incr/decr){
//code to be executed
}
```




```
#include <iostream.h>
void main() {
        for(int i=1;i<=10;i++){
          cout<<i <<"\n";
         }
    }
```

Output:

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

C++ Nested For Loop Example

Let's see a simple example of nested for loop in C++.

```
#include <iostream.h>
void main () {
    for(int i=1;i<=3;i++){
        for(int j=1;j<=3;j++){
        cout<<i<<" "<<j<<"\n";
      }
    }
}
```

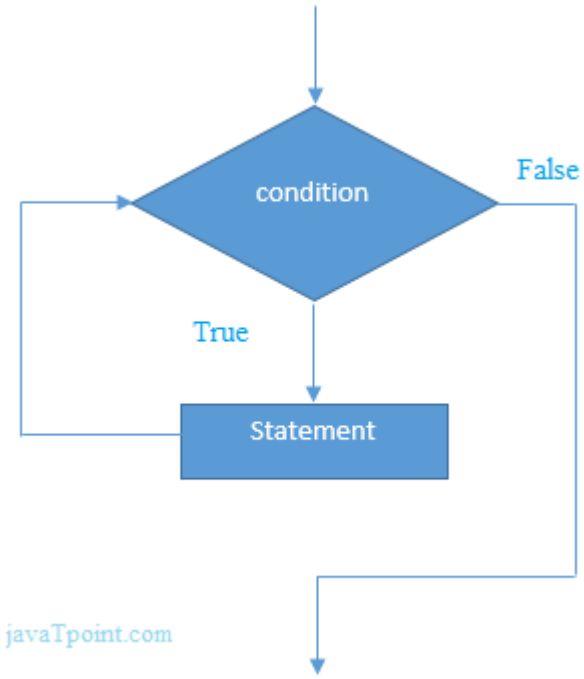
Output:

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

C++ While loop

In C++, while loop is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop than for loop.

```
while(condition){
//code to be executed
}
```



C++ While Loop Example

Let's see a simple example of while loop to print table of 1.

```
#include <iostream.h>
void main() {
 int i=1;
       while(i<=10)
      {
          cout<<i <<"\n";
          i++;
        }
   }
```

Output:

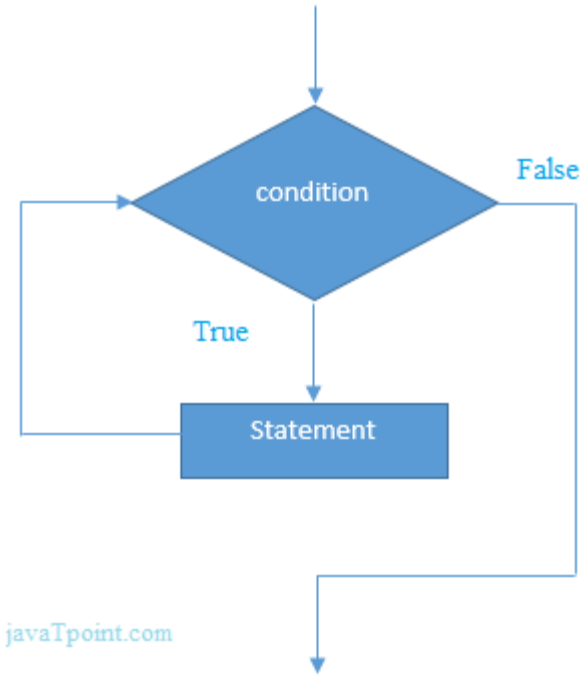
```
1
2
3
4
5
6
7
8
9
10
```

C++ Do-While Loop

The C++ do-while loop is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.

The C++ do-while loop is executed at least once because condition is checked after loop body.

```
do{
//code to be executed
}while(condition);
```



C++ do-while Loop Example

Let's see a simple example of C++ do-while loop to print the table of 1.

```
#include <iostream.h>
void main() {
    int i = 1;
        do{
            cout<<i<<"\n";
            i++;
        } while (i <= 10) ;
}
```

Output:

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

C++ Break Statement

The C++ break is used to break loop or switch statement. It breaks the current flow of the program at the given condition. In case of inner loop, it breaks only inner loop.

```
jump-statement;
```

```
break;
```

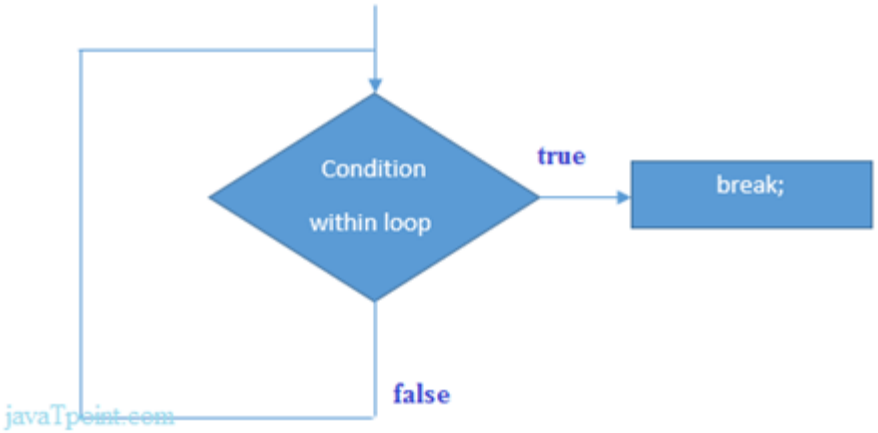


Figure: Flowchart of break statement

C++ Break Statement Example

Let's see a simple example of C++ break statement which is used inside the loop.

```
#include <iostream.h>
void main() {
    for (int i = 1; i <= 10; i++)
       {
          if (i == 5)
          {
             break;
          }
      cout<<i<<"\n";
       }
}
```

Output:

```
1
2
3
4
```

C++ Continue Statement

The C++ continue statement is used to continue loop. It continues the current flow of the program and skips the remaining code at specified condition. In case of inner loop, it continues only inner loop.

```
jump-statement;
continue;
```

C++ Continue Statement Example

```
#include <iostream.h>
void main()
{
    for(int i=1;i<=10;i++){
        if(i==5){
            continue;
        }
        cout<<i<<"\n";
     }
}
```

Output:

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

C++ Goto Statement

The C++ goto statement is also known as jump statement. It is used to transfer control to the other part of the program. It unconditionally jumps to the specified label.

It can be used to transfer control from deeply nested loop or switch case label.

C++ Goto Statement Example

Let's see the simple example of goto statement in C++.

```
#include <iostream.h>
void main()
{
ineligible:
      cout<<"You are not eligible to vote!\n";
    cout<<"Enter your age:\n";
    int age;
    cin>>age;
    if (age < 18){
        goto ineligible;
    }
    else
    {
        cout<<"You are eligible to vote!";
    }
```

```
}
```

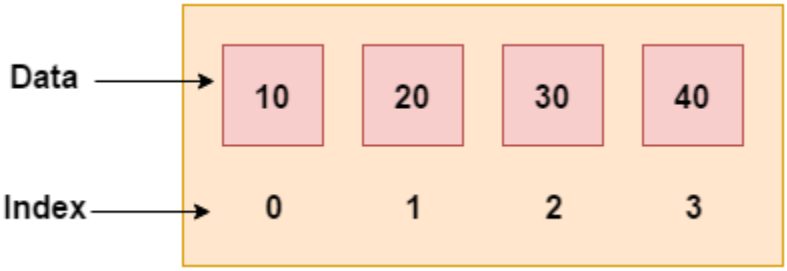
Output:

```
You are not eligible to vote!
Enter your age:
16
You are not eligible to vote!
Enter your age:
7
You are not eligible to vote!
Enter your age:
22
You are eligible to vote!
```

C++ Arrays

Like other programming languages, array in C++ is a group of similar types of elements that have contiguous memory location.

In C++ **std::array** is a container that encapsulates fixed size arrays. In C++, array index starts from 0. We can store only fixed set of elements in C++ array.



Advantages of C++ Array

- Code Optimization (less code)
- Random Access
- Easy to traverse data
- Easy to manipulate data
- Easy to sort data etc.

Disadvantages of C++ Array

- Fixed size

C++ Array Types

There are 2 types of arrays in C++ programming:

1. Single Dimensional Array
2. Multidimensional Array

C++ Single Dimensional Array

Let's see a simple example of C++ array, where we are going to create, initialize and traverse array.

```
#include <iostream>
void main()
{
 int arr[5]={10, 0, 20, 0, 30};  //creating and initializing array
       //traversing array
       for (int i = 0; i < 5; i++)
       {
          cout<<arr[i]<<"\n";
       }
}
```

Output:

```
10
0
20
0
30
```

C++ Array Example: Traversal using foreach loop

We can also traverse the array elements using foreach loop. It returns array element one by one.

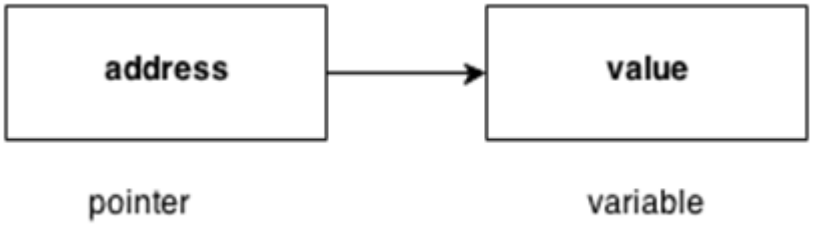
```
#include <iostream.h>
void main()
{
 int arr[5]={10, 0, 20, 0, 30}; //creating and initializing array
       //traversing array
      for (int i: arr)
       {
          cout<<i<<"\n";
       }
}
```

Output:

```
10
20
30
40
50
```

C++ Pointers

The pointer in C++ language is a variable, it is also known as locator or indicator that points to an address of a value.



Advantage of pointer

1) Pointer reduces the code and improves the performance, it is used to retrieving strings, trees etc. and used with arrays, structures and functions.

2) We can return multiple values from function using pointer.

3) It makes you able to access any memory location in the computer's memory.

Usage of pointer

There are many usage of pointers in C++ language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where pointer is used.

2) Arrays, Functions and Structures

Pointers in c language are widely used in arrays, functions and structures. It reduces the code and improves the performance.

Symbols used in pointer

Symbol	Name	Description
& (ampersand sign)	Address operator	Determine the address of a variable.
* (asterisk sign)	Indirection operator	Access the value of an address.

Declaring a pointer

The pointer in C++ language can be declared using * (asterisk symbol).

1. **int** * a; //pointer to int
2. **char** * c; //pointer to char

Pointer Example

Let's see the simple example of using pointers printing the address and value.

```
#include <iostream.h>
int main()
{
int number=30;
int * p;
p=&number;//stores the address of number variable
cout<<"Address of number variable is:"<<&number<<endl;
cout<<"Address of p variable is:"<<p<<endl;
cout<<"Value of p variable is:"<<*p<<endl;
  return 0;
}
```

Output:

```
Address of number variable is:0x7ffccc8724c4
Address of p variable is:0x7ffccc8724c4
Value of p variable is:30
```

Pointer Program to swap 2 numbers without using 3rd variable

```
#include <iostream.h>
int main()
{
int a=20,b=10,*p1=&a,*p2=&b;
cout<<"Before swap: *p1="<<*p1<<" *p2="<<*p2<<endl;
*p1=*p1+*p2;
*p2=*p1-*p2;
*p1=*p1-*p2;
cout<<"After swap: *p1="<<*p1<<" *p2="<<*p2<<endl;
  return 0;
}
```

Output:

```
Before swap: *p1=20 *p2=10
After swap: *p1=10 *p2=20
```

C++ Strings

In C++, string is an object of **std::string** class that represents sequence of characters. We can perform many operations on strings such as concatenation, comparison, conversion etc.

C++ String Example

Let's see the simple example of C++ string.

```
#include <iostream>
using namespace std;
int main( ) {
   string s1 = "Hello";
      char ch[] = { 'C', '+', '+'};
      string s2 = string(ch);
      cout<<s1<<endl;
      cout<<s2<<endl;
}
```

Output:

```
Hello
C++
```

C++ String Compare Example

Let's see the simple example of string comparison using strcmp() function.

```
#include <iostream>
#include <cstring>
using namespace std;
int main ()
{
  char key[] = "mango";
  char buffer[50];
  do {
     cout<<"What is my favourite fruit? ";
     cin>>buffer;
  } while (strcmp (key,buffer) != 0);
 cout<<"Answer is correct!!"<<endl;
  return 0;
}
```

Output:

```
What is my favourite fruit? apple
What is my favourite fruit? banana
What is my favourite fruit? mango
Answer is correct!!
```

C++ String Concat Example

Let's see the simple example of string concatenation using strcat() function.

```
#include <iostream>
```

```
#include <cstring>
using namespace std;
int main()
{
   char key[25], buffer[25];
   cout << "Enter the key string: ";
   cin.getline(key, 25);
   cout << "Enter the buffer string: ";
    cin.getline(buffer, 25);
   strcat(key, buffer);
   cout << "Key = " << key << endl;
   cout << "Buffer = " << buffer<<endl;
   return 0;
}
```

Output:

```
Enter the key string: Welcome to
Enter the buffer string:  C++ Programming.
Key = Welcome to C++ Programming.
Buffer =  C++ Programming.
```

C++ String Copy Example

Let's see the simple example of copy the string using strcpy() function.

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
   char key[25], buffer[25];
   cout << "Enter the key string: ";
   cin.getline(key, 25);
   strcpy(buffer, key);
   cout << "Key = "<< key << endl;
   cout << "Buffer = "<< buffer<<endl;
   return 0;
}
```

Output:

```
Enter the key string: C++ Tutorial
Key = C++ Tutorial
Buffer = C++ Tutorial
```

C++ String Length Example

Let's see the simple example of finding the string length using strlen() function.

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
   char ary[] = "Welcome to C++ Programming";
   cout << "Length of String = " << strlen(ary)<<endl;
   return 0;
}
```

Output:

```
Length of String = 26
```

C++ String Functions

Function	Description
int compare(const string& str)	It is used to compare two string objects.
int length()	It is used to find the length of the string.
void swap(string& str)	It is used to swap the values of two string objects.
string substr(int pos,int n)	It creates a new string object of n characters.
int size()	It returns the length of the string in terms of bytes.
void resize(int n)	It is used to resize the length of the string up to n characters.
string& replace(int pos,int len,string& str)	It replaces portion of the string that begins at character position pos and spans len characters.
string& append(const string& str)	It adds new characters at the end of another string object.
char& at(int pos)	It is used to access an individual character at specified position pos.
int find(string& str,int pos,int n)	It is used to find the string specified in the parameter.
int find_first_of(string& str,int pos,int n)	It is used to find the first occurrence of the specified sequence.
int find_first_not_of(string&	It is used to search the string for the first character that does

str,int pos,int n)	not match with any of the characters specified in the string.
int find_last_of(string& str,int pos,int n)	It is used to search the string for the last character of specified sequence.
int find_last_not_of(string& str,int pos)	It searches for the last character that does not match with the specified sequence.
string& insert()	It inserts a new character before the character indicated by the position pos.
int max_size()	It finds the maximum length of the string.
void push_back(char ch)	It adds a new character ch at the end of the string.
void pop_back()	It removes a last character of the string.
string& assign()	It assigns new value to the string.
int copy(string& str)	It copies the contents of string into another.
char& back()	It returns the reference of last character.
Iterator begin()	It returns the reference of first character.
int capacity()	It returns the allocated space for the string.
const_iterator cbegin()	It points to the first element of the string.
const_iterator cend()	It points to the last element of the string.
void clear()	It removes all the elements from the string.
const_reverse_iterator crbegin()	It points to the last character of the string.
const_char* data()	It copies the characters of string into an array.
bool empty()	It checks whether the string is empty or not.
string& erase()	It removes the characters as specified.
char& front()	It returns a reference of the first character.
string& operator+=()	It appends a new character at the end of the string.
string& operator=()	It assigns a new value to the string.

char operator[](pos)	It retrieves a character at specified position pos.
int rfind()	It searches for the last occurrence of the string.
iterator end()	It references the last character of the string.
reverse_iterator rend()	It points to the first character of the string.
void shrink_to_fit()	It reduces the capacity and makes it equal to the size of the string.
char* c_str()	It returns pointer to an array that contains null terminated sequence of characters.
const_reverse_iterator crend()	It references the first character of the string.
reverse_iterator rbegin()	It reference the last character of the string.
void reserve(inr len)	It requests a change in capacity.
allocator_type get_allocator();	It returns the allocated object associated with the string.

C++ Functions

The function in C++ language is also known as procedure or subroutine in other programming languages.

To perform any task, we can create function. A function can be called many times. It provides modularity and code reusability.

Advantage of functions in C

There are many advantages of functions.

1) Code Reusability

By creating functions in C++, you can call it many times. So we don't need to write the same code again and again.

2) Code optimization

It makes the code optimized, we don't need to write much code.

Suppose, you have to check 3 numbers (531, 883 and 781) whether it is prime number or not. Without using function, you need to write the prime number logic 3 times. So, there is repetition of code.

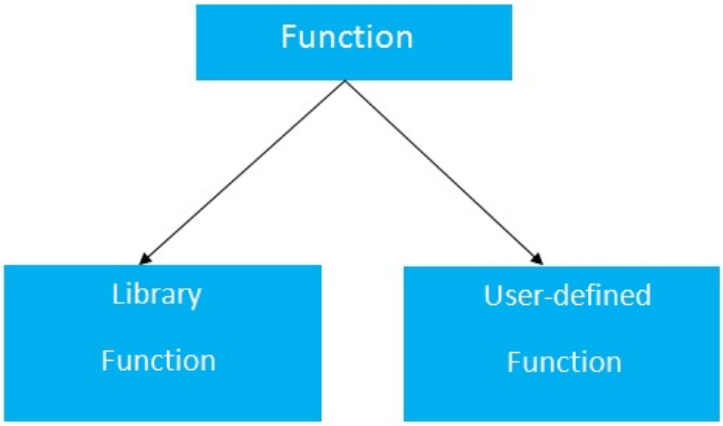
But if you use functions, you need to write the logic only once and you can reuse it several times.

Types of Functions

There are two types of functions in C programming:

1. Library Functions: are the functions which are declared in the C++ header files such as ceil(x), cos(x), exp(x), etc.

2. User-defined functions: are the functions which are created by the C++ programmer, so that he/she can use it many times. It reduces complexity of a big program and optimizes the code.



Declaration of a function

The syntax of creating function in C++ language is given below:

```
return_type function_name(data_type parameter...)
{
//code to be executed
}
```

C++ Function Example

Let's see the simple example of C++ function.

```
#include <iostream.h>
void func() {
   static int i=0; //static variable
   int j=0; //local variable
   i++;
   j++;
   cout<<"i=" << i<<" and j=" <<j<<endl;
}
void main()
{
 func();
 func();
```

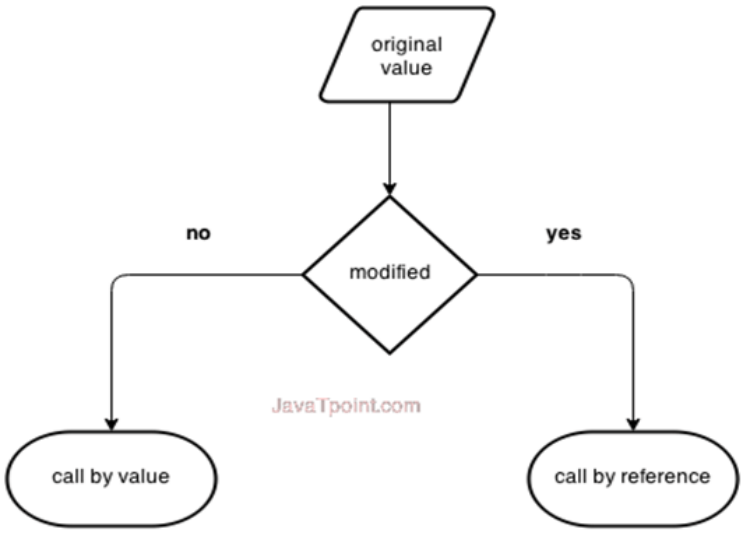
```
 func();
}
```

Output:

```
i= 1 and j= 1
i= 2 and j= 1
i= 3 and j= 1
```

Call by value and call by reference in C++

There are two ways to pass value or data to function in C language: call by value and call by reference. Original value is not modified in call by value but it is modified in call by reference.



Let's understand call by value and call by reference in C++ language one by one.

Call by value in C++

In call by value, **original value is not modified.**

In call by value, value being passed to the function is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only. It will not change the value of variable inside the caller method such as main().

Let's try to understand the concept of call by value in C++ language by the example given below:

```
#include <iostream.h>
void change(int data);
int main()
{
int data = 3;
change(data);
cout << "Value of the data is: " << data<< endl;
return 0;
}
```

```
void change(int data)
{
data = 5;
}
```

Output:

```
Value of the data is: 3
```

Call by reference in C++

In call by reference, original value is modified because we pass reference (address).

Here, address of the value is passed in the function, so actual and formal arguments share the same address space. Hence, value changed inside the function, is reflected inside as well as outside the function.

Note: To understand the call by reference, you must have the basic knowledge of pointers.

Let's try to understand the concept of call by reference in C++ language by the example given below:

```
#include<iostream.h>
void swap(int *x, int *y)
{
 int swap;
 swap=*x;
 *x=*y;
 *y=swap;
}
int main()
{
 int x=500, y=100;
 swap(&x, &y);  // passing value to function
 cout<<"Value of x is: "<<x<<endl;
 cout<<"Value of y is: "<<y<<endl;
 return 0;
}
```

Output:

```
Value of x is: 100
Value of y is: 500
```

Difference between call by value and call by reference in C++

No.	Call by value	Call by reference

1	A copy of value is passed to the function	An address of value is passed to the function
2	Changes made inside the function is not reflected on other functions	Changes made inside the function is reflected outside the function also
3	Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location

C++ Recursion

When function is called within the same function, it is known as recursion in C++. The function which calls the same function, is known as recursive function.

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

Let's see a simple example of recursion.

```
recursionfunction(){
recursionfunction(); //calling self function
}
```

C++ Recursion Example

Let's see an example to print factorial number using recursion in C++ language.

```
#include<iostream.h>
void main()
{
int factorial(int);
int fact,value;
cout<<"Enter any number: ";
cin>>value;
fact=factorial(value);
cout<<"Factorial of a number is: "<<fact<<endl;
}
int factorial(int n)
{
if(n<0)
return(-1); /*Wrong value*/
if(n==0)
return(1);  /*Terminating condition*/
else
{
return(n*factorial(n-1));
```

```
}
}
```

Output:

```
Enter any number: 5
Factorial of a number is: 120
```

We can understand the above program of recursive method call by the figure given below:

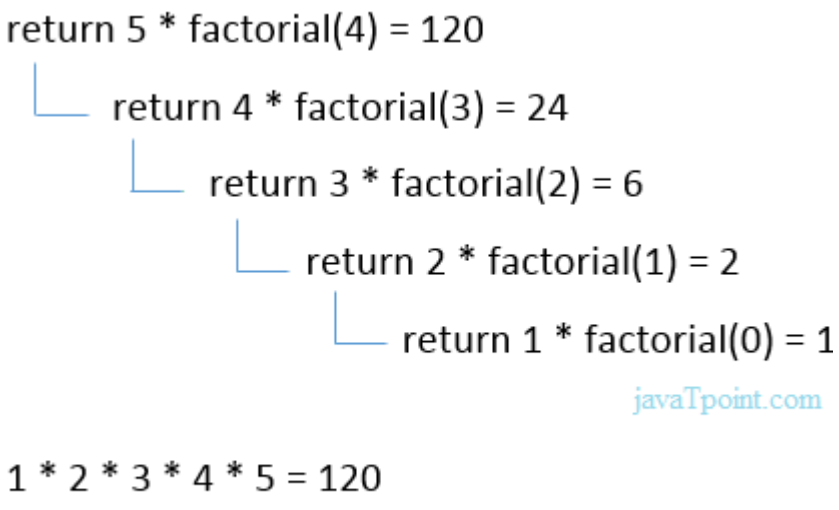


Fig: Recursion

C++ Storage Classes

Storage class is used to define the lifetime and visibility of a variable and/or function within a C++ program.

Lifetime refers to the period during which the variable remains active and visibility refers to the module of a program in which the variable is accessible.

There are five types of storage classes, which can be used in a C++ program

1. Automatic
2. Register
3. Static
4. External
5. Mutable

Storage Class	Keyword	Lifetime	Visibility	Initial Value
Automatic	auto	Function Block	Local	Garbage
Register	register	Function Block	Local	Garbage
Mutable	mutable	Class	Local	Garbage

External	extern	Whole Program	Global	Zero
Static	static	Whole Program	Local	Zero

Automatic Storage Class

It is the default storage class for all local variables. The auto keyword is applied to all local variables automatically.

```
auto int y;
float y = 3.45;
```

The above example defines two variables with a same storage class, auto can only be used within functions.

Register Storage Class

The register variable allocates memory in register than RAM. Its size is same of register size. It has a faster access than other variables.

It is recommended to use register variable only for quick access such as in counter.

Note: We can't get the address of register variable.

```
register int counter=0;
```

Static Storage Class

The static variable is initialized only once and exists till the end of a program. It retains its value between multiple functions call.

The static variable has the default value 0 which is provided by compiler.

```
#include <iostream.h>
void func() {
  static int i=0; //static variable
  int j=0; //local variable
  i++;
  j++;
  cout<<"i=" << i<<" and j=" <<j<<endl;
}
int main()
{
 func();
 func();
 func();
```

```
}
```

Output:

```
i= 1 and j= 1
i= 2 and j= 1
i= 3 and j= 1
```

External Storage Class

The extern variable is visible to all the programs. It is used if two or more files are sharing same variable or function.

```
extern int counter=0;
```

inline function: C++ **inline** function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.

To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.

A function definition in a class definition is an inline function definition, even without the use of the **inline** specifier.

Following is an example, which makes use of inline function to return max of two numbers −

Live Demo

```
#include <iostream.h>

inline int Max(int x, int y) {
   return (x > y)? x : y;
}

// Main function for the program
int main() {
   cout << "Max (20,10): " << Max(20,10) << endl;
   cout << "Max (0,200): " << Max(0,200) << endl;
   cout << "Max (100,1010): " << Max(100,1010) << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Max (20,10): 20
Max (0,200): 200
Max (100,1010): 1010
```

What is Memory Management?

Memory management is a process of managing computer memory, assigning the memory space to the programs to improve the overall system performance.

Why is memory management required?

As we know that arrays store the homogeneous data, so most of the time, memory is allocated to the array at the declaration time. Sometimes the situation arises when the exact memory is not determined until runtime. To avoid such a situation, we declare an array with a maximum size, but some memory will be unused. To avoid the wastage of memory, we use the new operator to allocate the memory dynamically at the run time.

Memory Management Operators

In C language, we use the **malloc()** or **calloc()** functions to allocate the memory dynamically at run time, and free() function is used to deallocate the dynamically allocated memory. C++ also supports these functions, but C++ also defines unary operators such as **new** and **delete** to perform the same tasks, i.e., allocating and freeing the memory.

New operator

A **new** operator is used to create the object while a **delete** operator is used to delete the object. When the object is created by using the new operator, then the object will exist until we explicitly use the delete operator to delete the object. Therefore, we can say that the lifetime of the object is not related to the block structure of the program.

Syntax

```
pointer_variable = new data-type
```

The above syntax is used to create the object using the new operator. In the above syntax, **'pointer variable'** is the name of the pointer variable, **'new'** is the operator, and **'data-type'** defines the type of the data.

Example 1:

```
int *p;
p = new int;
```

In the above example, 'p' is a pointer of type int.

Example 2:

```
float *q;
q = new float;
```

In the above example, 'q' is a pointer of type float.

In the above case, the declaration of pointers and their assignments are done separately. We can also combine these two statements as follows:

```
int *p = new int;
float *q =  new float;
```

Assigning a value to the newly created object

Two ways of assigning values to the newly created object:

- We can assign the value to the newly created object by simply using the assignment operator. In the above case, we have created two pointers 'p' and 'q' of type int and float, respectively. Now, we assign the values as follows:

  ```
  *p = 45;
  *q = 9.8;
  ```

We assign 45 to the newly created int object and 9.8 to the newly created float object.

- We can also assign the values by using new operator which can be done as follows:

  ```
  pointer_variable = new data-type(value);
  ```

Let's look at some examples.

1. **int** *p = **new int**(45);
2. **float** *p = **new float**(9.8);

How to create a single dimensional array

As we know that new operator is used to create memory space for any data-type or even user-defined data type such as an array, structures, unions, etc., so the syntax for creating a one-dimensional array is given below:

```
pointer-variable = new data-type[size];
```

Examples:

```
int *a1 = new int[8];
```

In the above statement, we have created an array of type int having a size equal to 8 where p[0] refers first element, p[1] refers the first element, and so on.

Delete operator

When memory is no longer required, then it needs to be deallocated so that the memory can be used for another purpose. This can be achieved by using the delete operator, as shown below:

```
delete pointer_variable;
```

In the above statement, **'delete'** is the operator used to delete the existing object,
and **'pointer_variable'** is the name of the pointer variable.

In the previous case, we have created two pointers 'p' and 'q' by using the new operator, and can be deleted by using the following statements:

```
delete p;
delete q;
```

The dynamically allocated array can also be removed from the memory space by using the following syntax:

```
delete [size] pointer_variable;
```

In the above statement, we need to specify the size that defines the number of elements that are required to be freed. The drawback of this syntax is that we need to remember the size of the array. But, in recent versions of C++, we do not need to mention the size as follows:

```
delete [ ] pointer_variable;
```

Let's understand through a simple example:

```
#include <iostream.h>
int main()
{
int size;  // variable declaration
int *arr = new int[size];   // creating an array
cout<<"Enter the size of the array : ";
std::cin >> size;    //
cout<<"\nEnter the element : ";
for(int i=0;i<size;i++)   // for loop
{
cin>>arr[i];
}
cout<<"\nThe elements that you have entered are :";
for(int i=0;i<size;i++)    // for loop
{
cout<<arr[i]<<",";
}
delete arr;  // deleting an existing array.
return 0;
}
```

In the above code, we have created an array using the new operator. The above program will take the user input for the size of an array at the run time. When the program completes all the operations, then it deletes the object by using the statement **delete arr**.

Output

```
Enter the size of the array : 5

Enter the element : 1
2
3
4
5

The elements that you have entered are :1,2,3,4,5,

...Program finished with exit code 0
Press ENTER to exit console.
```

Advantages of the new operator

The following are the advantages of the new operator over malloc() function:

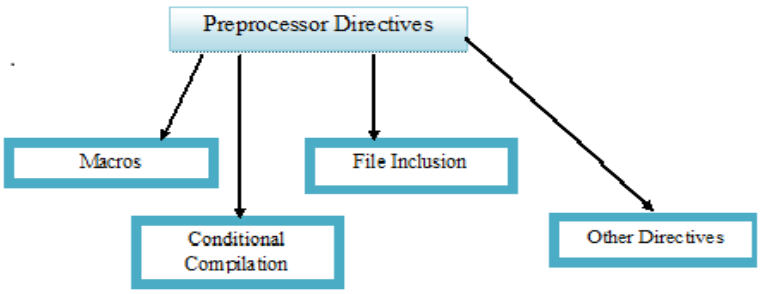
- It does not use the sizeof() operator as it automatically computes the size of the data object.
- It automatically returns the correct data type pointer, so it does not need to use the typecasting.
- Like other operators, the new and delete operator can also be overloaded.
- It also allows you to initialize the data object while creating the memory space for the object.

C++ Preprocessor Directive

WHAT ARE PREPROCESSOR DIRECTIVES?

- The preprocessor directives are the lines that are included in a program which begins with the # and it is different from the typical source code.
- They are invoked by the compiler to process the programs before compilation. The preprocessor directive is placed at the top of the source code in a separate line beginning with the character # and followed by a directive name.
- The preprocessor directive statement cannot be with semi colon. Some of the examples of preprocessor directives are #define, #include, #indef, etc.

TYPES OF PREPROCESSOR DIRECTIVES



MACROS

The macros are the piece of code in a program which is given some name and whenever this name occurs then the compiler replaces the name with the actual code. The #define directive is used to define a macro.

Let us have a look at the example to understand this:

```
#include <iostream>
  using namespace std;
// macro definition
#define range 10
int main()
{
int x;
{
   for (x = 0; x < range; x++)
{
       cout << x << "\n";
   }
   return 0;
}
}
```

We can also pass the arguments to macros. Macros defined with arguments work similarly as functions.

LET US HAVE A LOOK AT THE EXAMPLE:

```
#include <iostream>
  using namespace std;
// macro with parameter
#define AREA(l) (l * l)
int main()
{
   int x = 10, area;
     area = AREA(x);
     cout << "Area of square is: " << area;
     return 0;
}
```