Module 1

What is C++

C++ is a general purpose, case-sensitive, free-form programming language that supports object-oriented, procedural and generic programming.

C++ is a middle-level language, as it encapsulates both high and low level language features.

Object-Oriented Programming (OOPs)

C++ supports the object-oriented programming, the four major pillar of object-oriented programming (OOPs) used in C++ are:

1. Inheritance
2. Polymorphism
3. Encapsulation
4. Abstraction

C++ Standard Libraries

Standard C++ programming is divided into three important parts:

- The core library includes the data types, variables and literals, etc.
- The standard library includes the set of functions manipulating strings, files, etc.
- The Standard Template Library (STL) includes the set of methods manipulating a data structure.

Usage of C++

By the help of C++ programming language, we can develop different types of secured and robust applications:

- Window application

- Client-Server application
- Device drivers
- Embedded firmware etc

C++ Program

In this tutorial, all C++ programs are given with C++ compiler so that you can easily change the C++ program code.

File: main.cpp

```
1. #include <iostream>
2. using namespace std;
3. int main() {
4.    cout << "Hello C++ Programming";
5.    return 0;
6. }
```

A detailed explanation of first C++ program is given in next chapters.

C++ Program

Before starting the abcd of C++ language, you need to learn how to write, compile and run the first C++ program.

To write the first C++ program, open the C++ console and write the following code:

```
1. #include <iostream.h>
2. #include<conio.h>
3. void main() {
4.    clrscr();
5.    cout << "Welcome to C++ Programming.";
6.    getch();
7. }
```

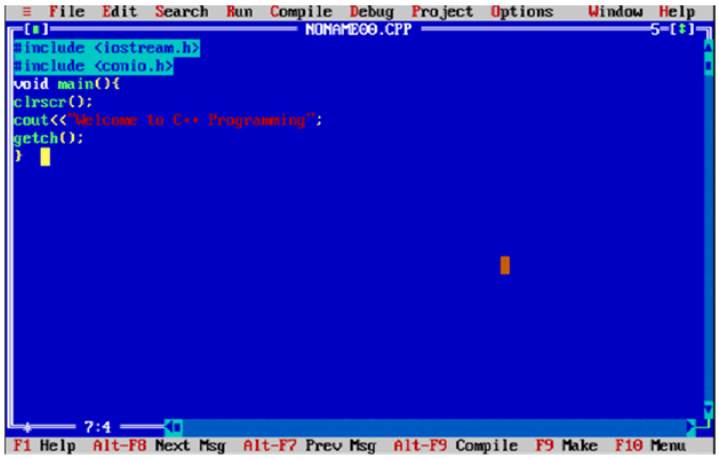
#include<iostream.h> includes the **standard input output** library functions. It provides **cin** and **cout** methods for reading from input and writing to output respectively.

#include <conio.h> includes the **console input output** library functions. The getch() function is defined in conio.h file.

void main() The **main() function is the entry point of every program** in C++ language. The void keyword specifies that it returns no value.

cout << "Welcome to C++ Programming." is **used to print the data "Welcome to C++ Programming."** on the console.

getch() The getch() function **asks for a single character**. Until you press any key, it blocks the screen.



How to compile and run the C++ program

There are 2 ways to compile and run the C++ program, by menu and by shortcut.

By menu

Now **click on the compile menu then compile sub menu** to compile the c++ program.

Then **click on the run menu then run sub menu** to run the c++ program.

By shortcut

Or, press ctrl+f9 keys compile and run the program directly.

You will see the following output on user screen.

You can view the user screen any time by pressing the **alt+f5** keys.

Now **press Esc** to return to the turbo c++ console.

C++ Variable

Variables are the fundamental building blocks of data manipulation and storage in programming, acting as ***dynamic containers*** for data in the C++ programming language. A ***variable*** is more than just a memory label. It serves as a link between abstract ideas and concrete data storage, allowing programmers to deftly manipulate data.

With the help of C++ variables, developers may complete a wide range of jobs, from simple ***arithmetic operations*** to complex algorithmic designs. These programmable containers can take on a variety of shapes, such as inte***gers, floating-point numbers, characters***, and ***user-defined structures***, each of which has a distinctive impact on the operation of the program.

Programmers follow a set of guidelines when generating variables, creating names that combine alphanumeric letters and underscores while avoiding reserved keywords. More than just placeholders, variables are what drive ***user input, intermediary calculations***, and the dynamic interactions that shape the program environment.

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:

1. type variable_list;

The example of declaring variable is given below:

1. **int** x;

```
2. float y;
3. 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:

```
1. int x=5,b=10;  //declaring 2 variable of integer type
2. float f=30.8;
3. 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:

```
1. int a;
2. int _ab;
3. int a30;
```

Invalid variable names:

```
1. int 4;
2. int x y;
3. int double;
```

Uses of C++ Variables:

There are several uses of variables in C++. Some main uses of C++ variables are as follows:

Important Ideas: Programming is fundamentally based on C++ variables, which allow for the ***storing, manipulation***, and ***interaction*** of data inside a program.

Memory Storage: Variables are named ***memory regions*** that may hold values of different data kinds, ranging from characters and integers to more intricate user-defined structures.

Dynamic character: Programming that is responsive and dynamic is made possible by the ability to ***assign, modify***, and ***reuse*** data through variables.

Data Types: The ***several data types*** that C++ provides, including ***int, float, char***, and others, each define the sort of value that a variable may store.

Variable declaration: Use the syntax ***type variable_name*** to define a variable, containing its ***type*** and ***name***.

Initialization: When a variable is declared, it can be given a value, such as ***int age = 25***.

Rules and Naming: Variable names must begin with a letter or an ***underscore***, avoid reserved ***keywords***, and be composed of ***letters, numbers***, and ***underscores***.

Utilization and Manipulation: The functionality of a program is improved by variables' participation in ***arithmetic, logical, and relational operations***.

Scope: Variables have a scope that specifies the areas of a program where they may be accessed and used.

Reusability and Modularity: Variables with appropriate names make code easier to ***comprehend, encourage modularity***, and allow for code reuse.

Object-Oriented: In object-oriented programming, variables are essential because they contain data inside of classes and objects.

Memory Control: Incorrect usage of variables can result in memory leaks or inefficient allocation, therefore understanding those helps with memory management.

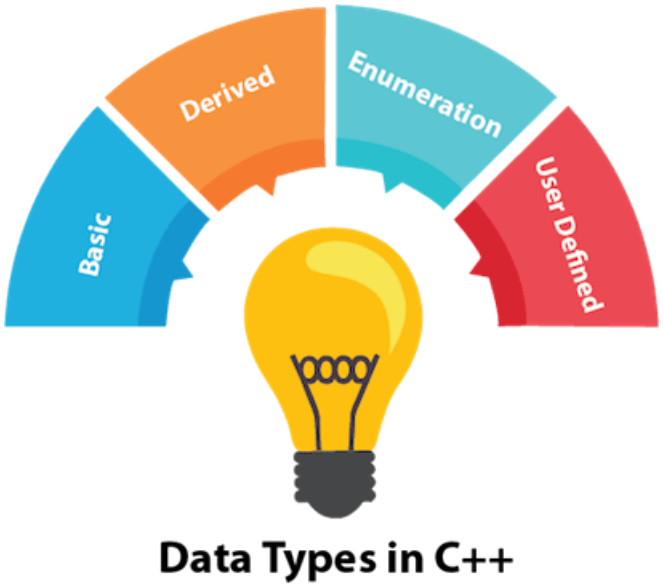
Applications in the Real World: Variables are used in a variety of applications, including ***web applications, system programming***, and ***scientific simulations***.

Debugging and upkeep: The proper use of variables reduces errors and enhances program quality while making debugging and code maintenance easier.

Interactivity: Variables are essential for interactive programs to capture user input and enable dynamic replies.

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	
signed long int	4 byte	
unsigned long int	4 byte	
float	4 byte	
double	8 byte	
long double	10 byte	

The ***nature*** and size of variables in C++ are heavily influenced by ***data types***, which also have an impact on ***memory use*** and the range of values that may be stored. Although the material given covers the fundamental data types, certain significant aspects and factors might improve your comprehension of C++ data types.

Types of Floating-Point Data:

C++ incorporates the idea of ***scientific notation*** for ***encoding floating-point*** literals in addition to ***float, double***, and ***long double***. In this system, exponentiation is denoted by the letter e or ***'E'***. For illustration:

```
1. float scientificNotation = 3.0e5; // Represents 300000.0
```

Fixed-Width Integer Types: ***C++11*** added ***fixed-width integer types*** to ensure consistent behavior across various platforms, which contain a set number of bits. These types, whose names include ***int8_t, uint16_t,*** and ***int32_t***, are specified in the ***cstdint*** header. Regardless of the underlying system, these types are particularly helpful when you want precise control over the size of numbers.

size of Operator: The ***sizeof operator*** is used to calculate a ***data type*** or variable's size (in bytes). For instance:

```
1. int sizeOfInt = sizeof(int);
```

Character and String Types: The C++ language uses the char ***data type*** to represent ***characters. Wide characters (wchar_t)*** for expanded character sets, and the ***char16_t*** and ***char32_t types*** for ***Unicode characters*** are also introduced by C++. The ***std::string*** class or character arrays (char or wchar_t) are used to represent strings.

References and Pointers: Despite the brief mention of derived ***data types, pointers,*** and ***references*** are crucial concepts in C++. A variable that stores the memory address of another variable is known as a ***pointer***. It permits the allocation and modification of ***dynamic memory***. On the other hand, ***references*** offer another method of accessing a variable by establishing an ***alias. Pointers*** and ***references*** are essential for complicated data structures and more sophisticated memory management.

Enumeration Data Type: An ***enumeration*** is a user-defined data type made up of named constants ***(enumerators)***, and it is specified by the ***enum data type. Enumerations*** are frequently used to increase the readability and maintainability of code by giving specified ***data names*** that make sense.

Structures and Classes as User-Defined Data Types: Although the ***struct*** was described in the example as a ***user-defined data type***, C++ also introduces the concept of classes. Classes provide you with the ability to build ***user-defined types*** that are more complicated and have ***member variables*** and ***related methods (functions)***. They serve as the foundation of C++'s object-oriented programming.

Bit Fields: C++ offers a method to declare how many bits should be used by each component of a structure. It is helpful when working with packed data structures or hardware registers.

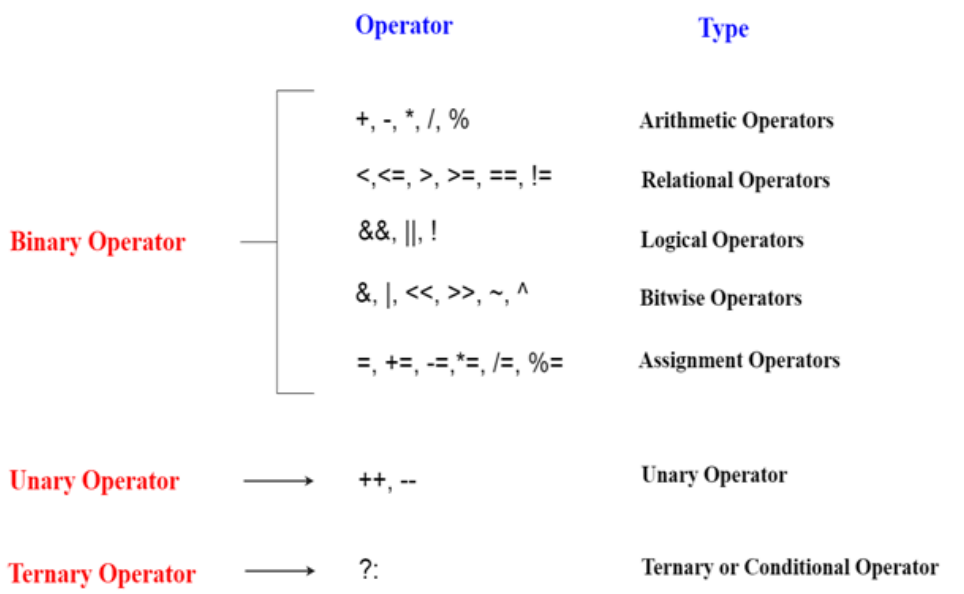
Data Type Modifiers: C++ supports the usage of ***data type modifiers*** like ***const, volatile,*** and ***mutable*** to modify the behavior of variables. For example, ***volatile*** denotes that a ***variable*** can be altered outside, but ***const*** makes a variable ***immutable***.

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

Header File	Function and Description
<iostream>	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

Let's see the simple example of standard output stream (cout):

```
1. #include <iostream>
2. using namespace std;
3. int main( ) {
4.    char ary[] = "Welcome to C++ tutorial";
5.    cout << "Value of ary is: " << ary << endl;
6. }
```

Output:

```
Value of ary is: Welcome to C++ tutorial
```

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):

```
1. #include <iostream>
2. using namespace std;
3. int main( ) {
4.   int age;
5.    cout << "Enter your age: ";
6.    cin >> age;
7.    cout << "Your age is: " << age << endl;
8. }
```

Output:

```
Enter your age: 22
Your age is: 22
```

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):

```
1. #include <iostream>
2. using namespace std;
3. int main( ) {
4. cout << "C++ Tutorial";
5. cout << "End of line"<<endl;
6. }
```

Output:

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

C++ if-else

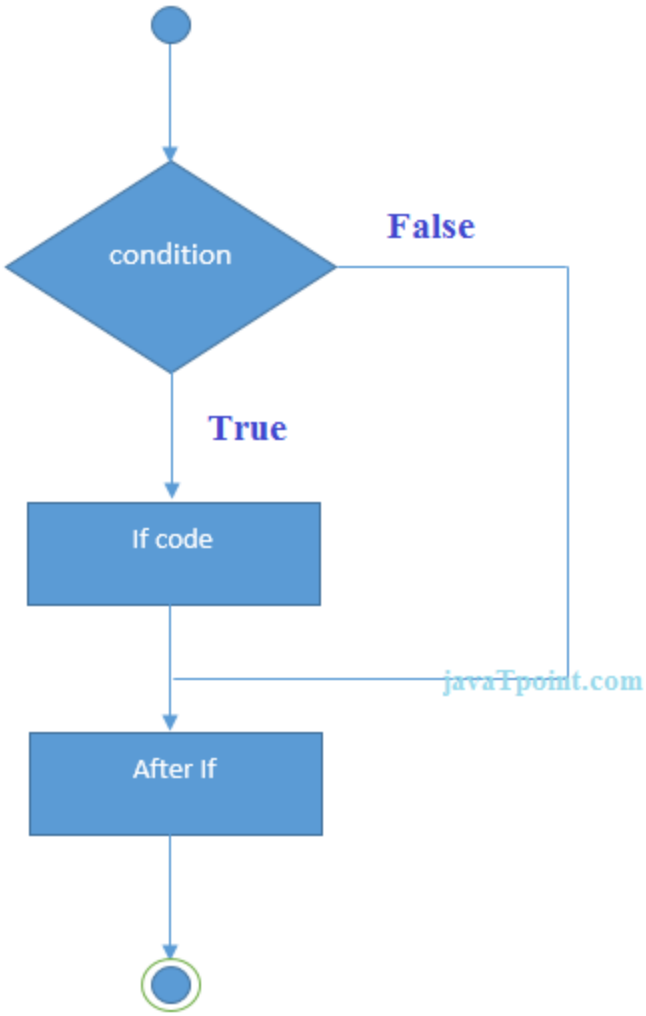
In C++ programming, if statement is used to test the condition. There are various types of if statements in C++.

- 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.

```
1. if(condition){
2. //code to be executed
3. }
```



C++ If Example

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main () {
5.    int num = 10;
6.          if (num % 2 == 0)
7.          {
8.             cout<<"It is even number";
9.          }
```

```
10.    return 0;
11. }
```

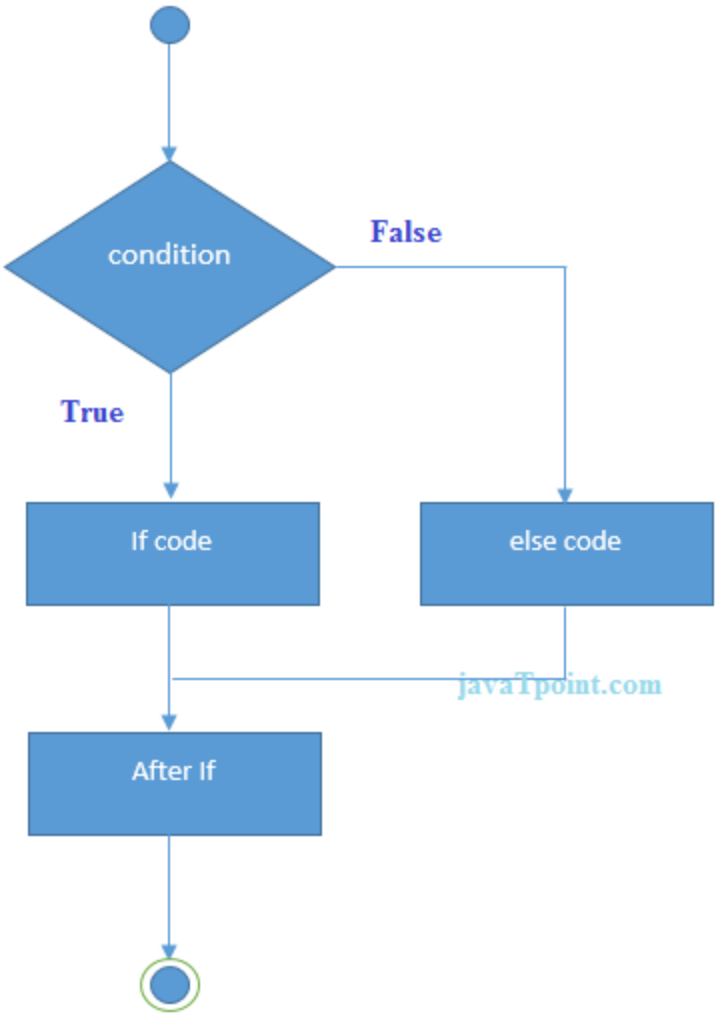
Output:/p>

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.

```
1. if(condition){
2. //code if condition is true
3. }else{
4. //code if condition is false
5. }
```



C++ If-else Example

```
1.  #include <iostream>
2.  using namespace std;
3.  int main () {
4.    int num = 11;
5.          if (num % 2 == 0)
6.          {
7.             cout<<"It is even number";
8.          }
9.          else
10.         {
```

```
11.             cout<<"It is odd number";
12.         }
13.   return 0;
14. }
```

Output:

It is odd number

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

```
1.  #include <iostream>
2.  using namespace std;
3.  int main () {
4.     int num;
5.     cout<<"Enter a Number: ";
6.     cin>>num;
7.          if (num % 2 == 0)
8.          {
9.             cout<<"It is even number"<<endl;
10.         }
11.         else
12.         {
13.            cout<<"It is odd number"<<endl;
14.         }
15.  return 0;
16. }
```

Output:

Enter a number:11
It is odd number

Output:

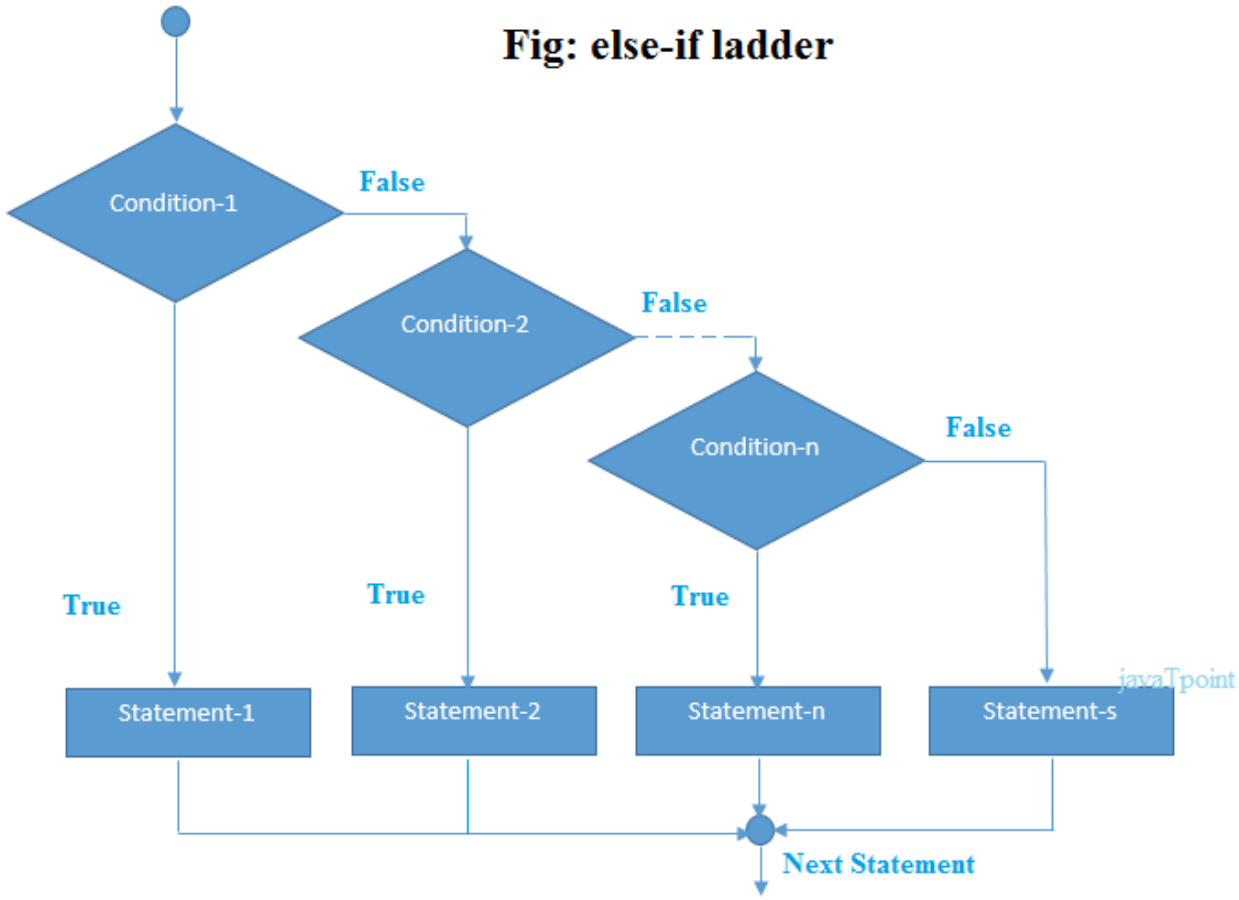
Enter a number:12

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
}
```



C++ If else-if Example

```
1.  #include <iostream>
2.  using namespace std;
3.  int main () {
4.      int num;
5.      cout<<"Enter a number to check grade:";
6.      cin>>num;
7.          if (num <0 || num >100)
8.          {
9.              cout<<"wrong number";
10.         }
11.         else if(num >= 0 && num < 50){
12.             cout<<"Fail";
```

```
13.         }
14.         else if (num >= 50 && num < 60)
15.         {
16.            cout<<"D Grade";
17.         }
18.         else if (num >= 60 && num < 70)
19.         {
20.            cout<<"C Grade";
21.         }
22.         else if (num >= 70 && num < 80)
23.         {
24.            cout<<"B Grade";
25.         }
26.         else if (num >= 80 && num < 90)
27.         {
28.            cout<<"A Grade";
29.         }
30.         else if (num >= 90 && num <= 100)
31.         {
32.            cout<<"A+ Grade";
33.         }
34.    }
```

Output:

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

Output:

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

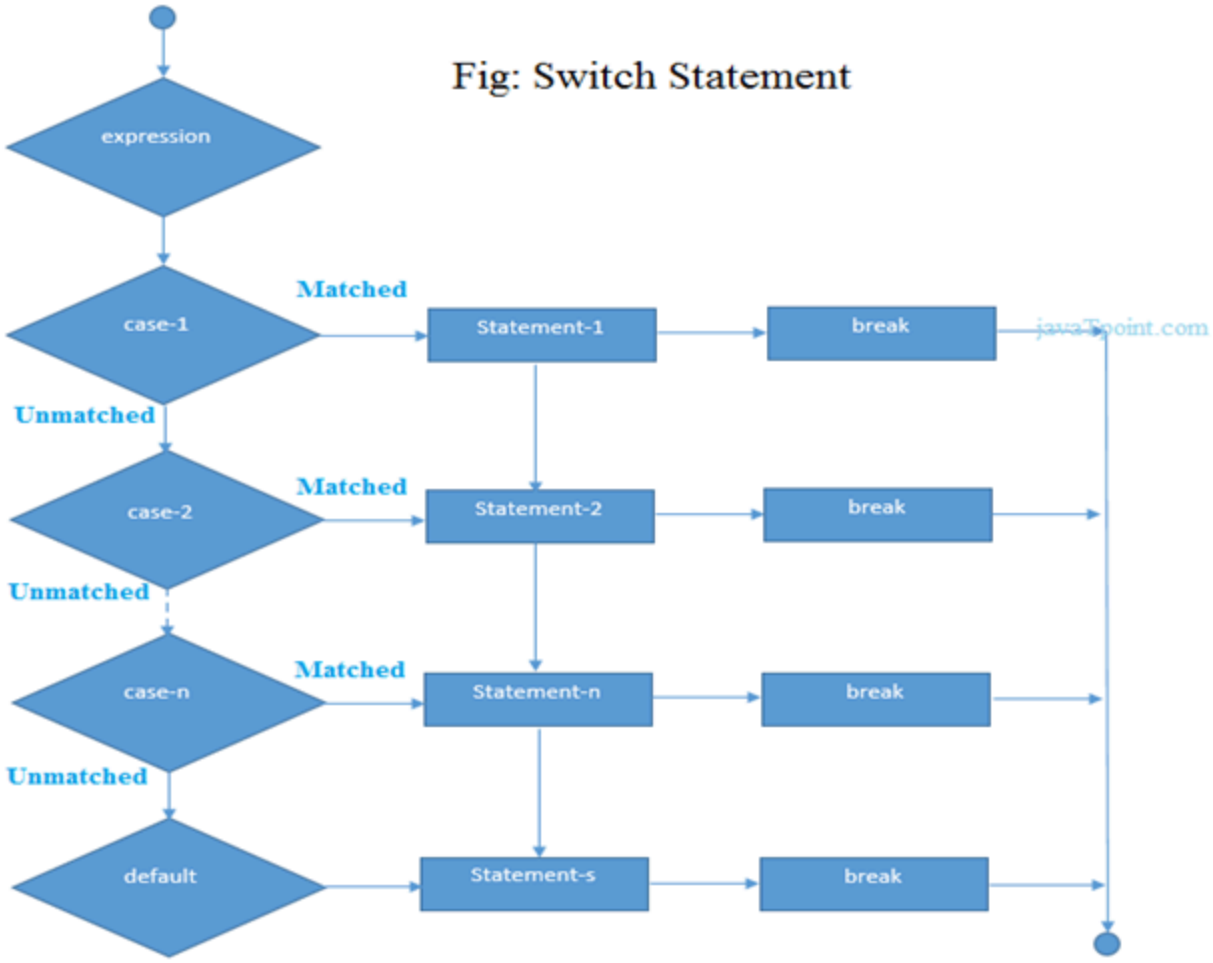
C++ switch

The ***switch statement*** in C++ is a potent ***control structure*** that enables you to run several code segments based on the result of an expression. It offers a sophisticated and effective substitute for utilizing a succession of ***if-else-if statements*** when you have to make a decision between several possibilities.

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;
}
```



C++ Switch Example

```
1.  #include <iostream>
2.  using namespace std;
3.  int main () {
4.        int num;
5.        cout<<"Enter a number to check grade:";
6.        cin>>num;
7.          switch (num)
8.         {
9.            case 10: cout<<"It is 10"; break;
10.           case 20: cout<<"It is 20"; break;
11.           case 30: cout<<"It is 30"; break;
```

```
12.             default: cout<<"Not 10, 20 or 30"; break;
13.         }
14.     }
```

Output:

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

Output:

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

Features of Switch Statement:

There are several features of the ***switch statement*** in C++. Some main features of the ***switch statement*** in C are as follows:

1. The ***fall-through*** behavior of the C++ ***switch statement*** is one of its key features. The control will ***fall through*** to the next case if a ***break statement*** is not used to ***stop*** a case block. After that, subsequent cases will be processed until a ***break*** is encountered or the end of the ***switch block*** is reached. This capability may be purposely used to share common code across several scenarios.
2. The ***switch statement's*** capacity to simplify code readability and maintenance is one of its fundamental advantages. Comparing a sequence of ***nested if-else statements*** to a ***switch statement*** when dealing with many situations can provide clearer, more organized code. Each case label gives the program a unique and unambiguous path to follow, improving the codebase's overall readability. It is very advantageous when working with extensive and complicated programs, where maintaining a ***logical flow***is crucial.
3. Another noteworthy benefit of the switch statement is ***efficiency***. When done correctly, a ***switch statement*** may frequently be more effective than a succession of ***if-else-if*** This

effectiveness results from the compiler's ability to optimize the switch statement to produce more effective machine code, which might lead to a quicker execution time. It's crucial to remember that the real speed improvements may differ based on the circumstance and compiler optimizations.

Limitations of Switch Statement

There are several limitations of the ***switch statement*** in C++. Some main limitations of the ***switch statement*** in C are as follows:

1. The ***switch statement*** has several restrictions, so it's important to be aware of those as well as industry standards. For instance, the ***switch statement's*** expression must be of the ***integral*** or ***enumeration type***. It limits its ability to be used with other data types like ***strings*** or ***floating-point integers***. Additionally, variables or expressions cannot be used as case labels since each case label must reflect a constant value that is known at ***compile-time***.
2. It is best practice to add a default case in the ***switch statement*** to guarantee thorough case coverage. Instances where none of the preceding instances match the value of the phrase are handled by this case. When none of the predetermined situations apply, including a ***default case*** prevents unexpected behavior and offers a clear path of action.

Conclusion:

In conclusion, the ***C++ switch statement*** is a flexible construct that makes it easier for programs to handle a variety of scenarios. Its explicit ***case labels*** and succinct syntax make code easier to comprehend and maintain, especially in situations when there are many possible outcomes. The ***switch statement*** improves the organization of program logic by offering a ***direct mapping*** between ***cases*** and ***actions***.

The ***switch statement*** has advantages over an ***if-else-if ladder*** in terms of performance since the compiler can optimize it for ***quicker execution***. Developers should be aware of its restrictions, such as the need for integral or enumeration expression types and constant case values.

It is advised to provide a default case in the ***switch statement*** to manage mismatched circumstances and use it efficiently and gently. Programmers may take advantage of the switch

statement's advantages to create better ***organized, effective***, and ***understandable*** C++ code by following best practices and comprehending its intricacies.

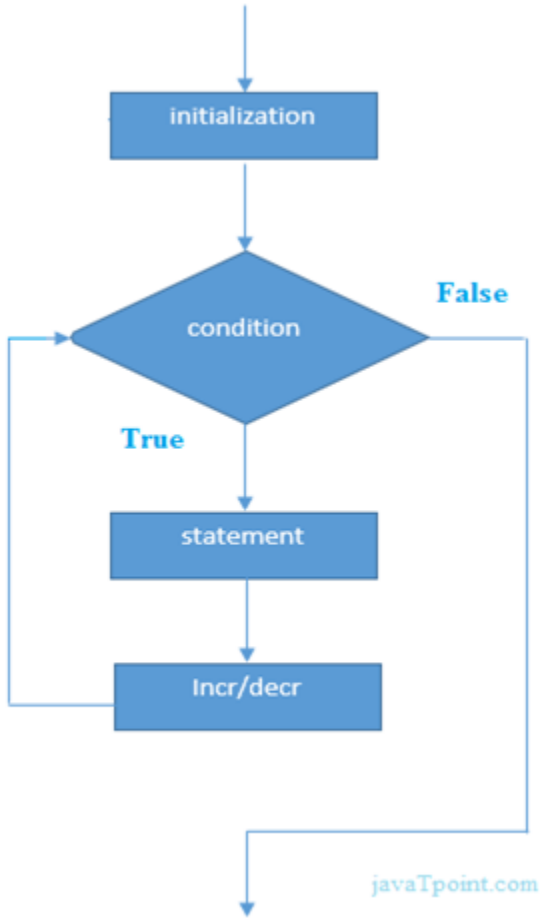
C++ For Loop

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.

```
1. for(initialization; condition; incr/decr){
2. //code to be executed
3. }
```

Flowchart:



C++ For Loop Example

```
1. #include <iostream>
2. using namespace std;
3. int main() {
4.         for(int i=1;i<=10;i++){
5.           cout<<i <<"\n";
6.         }
7.     }
```

Output:

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

C++ Nested For Loop

In C++, we can use for loop inside another for loop, it is known as nested for loop. The inner loop is executed fully when outer loop is executed one time. So if outer loop and inner loop are executed 4 times, inner loop will be executed 4 times for each outer loop i.e. total 16 times.

C++ Nested For Loop Example

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

```
1. #include <iostream>
2. using namespace std;
3.
```

```
4.  int main () {
5.      for(int i=1;i<=3;i++){
6.          for(int j=1;j<=3;j++){
7.          cout<<i<<" "<<j<<"\n";
8.       }
9.      }
10.  }
```

Output:

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

C++ Infinite For Loop

If we use double semicolon in for loop, it will be executed infinite times. Let's see a simple example of infinite for loop in C++.

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main () {
5.      for (; ;)
6.       {
7.            cout<<"Infinitive For Loop";
8.       }
9.   }
```

Output:

Infinitive For Loop
Infinitive For Loop
Infinitive For Loop
Infinitive For Loop
Infinitive For Loop
ctrl+c

Module 2

C++ OOPs Concepts

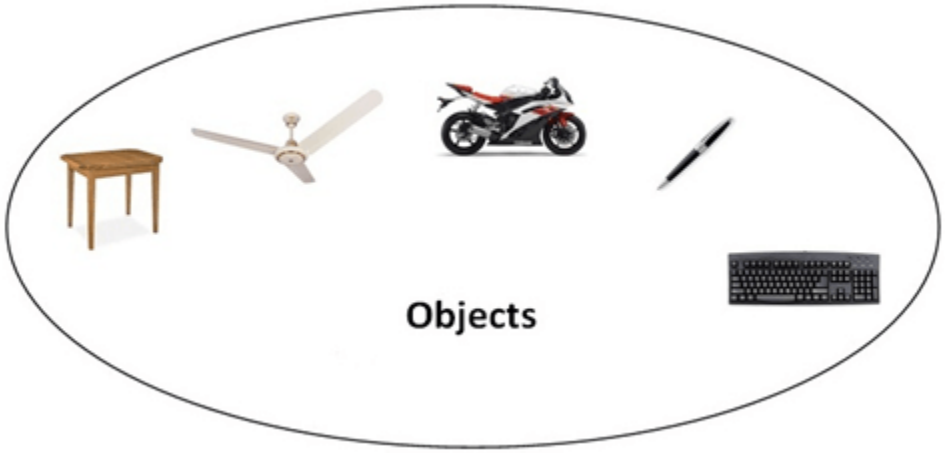
The major purpose of C++ programming is to introduce the concept of object orientation to the C programming language.

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.

A Class in C++ is the foundational element that leads to Object-Oriented programming. A class instance must be created in order to access and use the user-defined data type's data members and member functions. An object's class acts as its blueprint. Take the class of cars as an example. Even if different names and brands may be used for different cars, all of them will have some characteristics in common, such as four wheels, a speed limit, a range of miles, etc. In this case, the class of car is represented by the wheels, the speed limitations, and the mileage.

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.

1. Sub class - Subclass or Derived Class refers to a class that receives properties from another class.
2. Super class - The term "Base Class" or "Super Class" refers to the class from which a subclass inherits its properties.
3. Reusability - As a result, when we wish to create a new class, but an existing class already contains some of the code we need, we can generate our new class from the old class thanks to inheritance. This allows us to utilize the fields and methods of the pre-existing class.

Example:

1. class ABC : private XYZ //private derivation

{ }

2. class ABC : public XYZ //public derivation

{ }

3. class ABC : protected XYZ //protected derivation

{ }

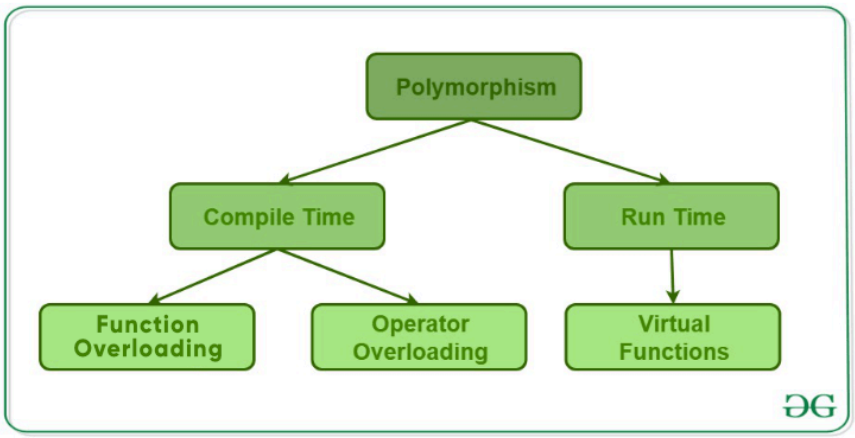
4. class ABC: XYZ //private derivation by default

{ }

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.

Different situations may cause an operation to behave differently. The type of data utilized in the operation determines the behaviour.



Compile-time polymorphism

It is defined as the polymorphism in which the function is called at the compile time. This type of process is also called *early or static binding*.

Examples of compile-time polymorphism

1) Function overloading

Function overloading is defined as using one function for different purposes. Here, one function performs many tasks by changing the function signature(number of arguments and types of arguments). It is an example of compile-time polymorphism because what function is to be called is decided at the time of compilation.

```
// C++ program to demonstrate

// function overloading or

// Compile-time Polymorphism

#include <bits/stdc++.h>

using namespace std;

class Geeks {

public:

        // Function with 1 int parameter

        void func(int x)
```

```
        {
                cout << "value of x is " << x << endl;
        }
        // Function with same name but
        // 1 double parameter
        void func(double x)
        {
                cout << "value of x is " << x << endl;
        }
        // Function with the same name and
        // 2 int parameters
        void func(int x, int y)
        {
                cout << "value of x and y is " << x << ", " << y<< endl;
        }
};
// Driver code
int main()
{
        Geeks obj1;
        // Function being called depends
        // on the parameters passed
        // func() is called with int value
        obj1.func(7);
```

```
        // func() is called with double value
        obj1.func(9.132);


        // func() is called with 2 int values
        obj1.func(85, 64);
        return 0;
}
```

Operator Overloading

Operator overloading is defined as using an operator for an addition operation besides the original one.

The concept of operator overloading is used as it provides special meanings to the user-defined data types. The benefit of operator overloading is we can use the same operand to serve two different operations. The basic operator overloading example is the '+' operator as it is used to add numbers and strings.

The operators **, :: ?: sizeof** cant be overloaded.

```
// C++ program to demonstrate
// Operator Overloading or
// Compile-Time Polymorphism
#include <iostream>
using namespace std;


class Complex {
private:
        int real, imag;
```

```
public:
	Complex(int r = 0, int i = 0)
	{
		real = r;
		imag = i;
	}
	// This is automatically called
	// when '+' is used with between
	// two Complex objects
	Complex operator+(Complex const& obj)
	{
		Complex res;
		res.real = real + obj.real;
		res.imag = imag + obj.imag;
		return res;
	}
	void print() { cout << real << " + i" << imag << endl; }
};
// Driver code
int main()
{	Complex c1(10, 5), c2(2, 4);
	// An example call to "operator+"
	Complex c3 = c1 + c2;
	c3.print();
```

```
}
```

Run-time polymorphism

The run-time polymorphism is defined as the process in which the function is called at the time of program execution.

An example of runtime polymorphism is *function overriding.*

```
// C++ program for function overriding with data members

#include <bits/stdc++.h>

using namespace std;

// base class declaration.

class Animal {

public:

        string color = "Black";

};

// inheriting Animal class.

class Dog : public Animal {

public:

        string color = "Grey";

};

// Driver code

int main(void)

{

        Animal d = Dog(); // accessing the field by reference

                                        // variable which refers to derived

        cout << d.color;
```

```
}
```

Abstraction

Hiding internal details and showing functionality is known as abstraction. Data abstraction is the process of exposing to the outside world only the information that is necessary while concealing implementation or background information. For example: phone call, we don't know the internal processing.

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

```
// C++ Program to Demonstrate the

// working of Abstraction

#include <iostream>

using namespace std;

class implementAbstraction {

private:

        int a, b;

public:

        // method to set values of

        // private members

        void set(int x, int y)

        {

                a = x;

                b = y;

        }

        void display()

        {

                cout << "a = " << a << endl;
```

```
        cout << "b = " << b << endl;
    }
};
int main()
{
    implementAbstraction obj;
    obj.set(10, 20);
    obj.display();
    return 0;
}
```

Encapsulation

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

Encapsulation is typically understood as the grouping of related pieces of information and data into a single entity. Encapsulation is the process of tying together data and the functions that work with it in object-oriented programming. Take a look at a practical illustration of encapsulation: at a company, there are various divisions, including the sales division, the finance division, and the accounts division. All financial transactions are handled by the finance sector, which also maintains records of all financial data. In a similar vein, the sales section is in charge of all tasks relating to sales and maintains a record of each sale. Now, a scenario could occur when, for some reason, a financial official requires all the information on sales for a specific month. Under the umbrella term "sales section," all of the employees who can influence the sales section's data are grouped together. Data abstraction or concealing is another side effect of encapsulation. In the same way that encapsulation hides the data. In the aforementioned example, any other area cannot access any of the data from any of the sections, such as sales, finance, or accounts.

Dynamic Binding - In dynamic binding, a decision is made at runtime regarding the code that will be run in response to a function call. For this, C++ supports virtual functions.

```
#include <iostream>

using namespace std;
```

```
class temp{
        int a;
int b;
Public:
int solve(int input){
        a=input;
        b=a/2;
        return b;
}
};
int main() {
int n;
cin>>n;
temp half;
int ans=half.solve(n);
cout<<ans<<endl;
}
```

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.

Why do we need oops in C++?

There were various drawbacks to the early methods of programming, as well as poor performance. The approach couldn't effectively address real-world issues since, similar to procedural-oriented programming, you couldn't reuse the code within the program again, there was a difficulty with global data access, and so on.

With the use of classes and objects, object-oriented programming makes code maintenance simple. Because inheritance allows for code reuse, the program is simpler because you don't have to write the same code repeatedly. Data hiding is also provided by ideas like encapsulation and abstraction.

Why is C++ a partial oop?

The object-oriented features of the C language were the primary motivation behind the construction of the C++ language.

The C++ programming language is categorised as a partial object-oriented programming language even though it supports OOP concepts, including classes, objects, inheritance, encapsulation, abstraction, and polymorphism.

1) The main function must always be outside the class in C++ and is required. This means that we may do without classes and objects and have a single main function in the application.

It is expressed as an object in this case, which is the first time Pure OOP has been violated.

2) Global variables are a feature of the C++ programming language that can be accessed by any other object within the program and are defined outside of it. Encapsulation is broken here. Even though C++ encourages encapsulation for classes and objects, it ignores it for global variables.

Overloading

Polymorphism also has a subset known as overloading. An existing operator or function is said to be overloaded when it is forced to operate on a new data type.

Friend Function in C++

As we already know that in an object-oriented programming language, the members of the class can only access the data and functions of the class but outside functions cannot access the data of the class. There might be situation that the outside functions need to access the private members of the class so in that case, friend function comes into the picture.

What is a friend function?

A friend function is a function of the class defined outside the class scope but it has the right to access all the private and protected members of the class.

The friend functions appear in the class definition but friends are the member functions.

Characteristics of a Friend function:

- The function is not in the scope of the class to which it has been declared as a friend.
- It cannot be called using the object as it is not in the scope of that class.
- It can be invoked like a normal function without using the object.
- It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
- It can be declared either in the private or the public part.

Why do we need a friend function in C++?

- Friend function in C++ is used when the class private data needs to be accessed directly without using object of that class.
- Friend functions are also used to perform operator overloading. As we already know about the function overloading, operators can also be overloaded with the help of operator overloading.

Characteristics of a Friend function

- The friend function is declared using friend keyword.
- It is not a member of the class but it is a friend of the class.
- As it is not a member of the class so it can be defined like a normal function.
- Friend functions do not access the class data members directly but they pass an object as an argument.
- It is like a normal function.
- If we want to share the multiple class's data in a function then we can use the friend function.

Syntax for the declaration of a friend function.

```
class class_name
{
    friend data_type function_name(argument/s);          // syntax of friend function
};
```

In the above declaration, the friend function is preceded by the keyword friend. The function can be defined anywhere in the program like a normal C++ function. The function definition does not use either the keyword **friend or scope resolution operator**.

Let's understand the friend function through an example.

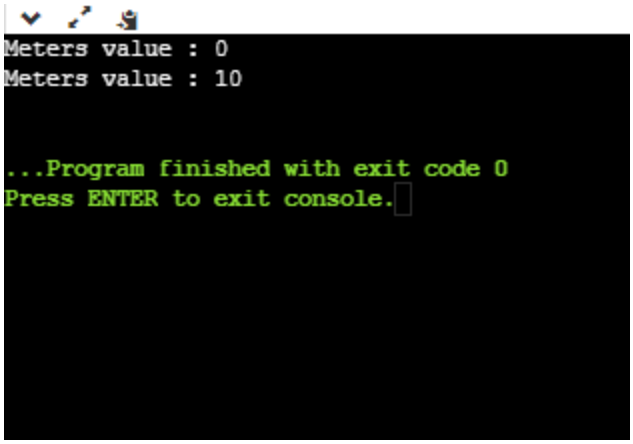
```
#include <iostream>
using namespace std;
class Distance
{
   private:
   int meters;
   public:
// constructor
```

```
    Distance()
    {
        meters = 0;
    }
    // definition of display_data() method
    void display_data()
    {
        std::cout << "Meters value : " << meters<<std::endl;
    }
    //prototype of a friend function.
    friend void addvalue(Distance &d);

};
// Definition of friend function
void addvalue(Distance &d) //  argument contain the reference
{
    d.meters = d.meters+10; // incrementing the value of meters by 10.
}
// main() method
int main()
{
    Distance d1; // creating the object of class distance.
    d1.display_data(); // meters = 0
    addvalue(d1); // calling friend function
    d1.display_data(); // meters = 10
    return 0;
}
```

In the above code, **Distance** is the class that contains private field named as '**meters**'. The **Distance()** is the constructor method that initializes the 'meters' value with 0. The **display_data()** is a method that displays the 'meters' value. The **addvalue()** is a friend function of Distance class that modifies the value of **'meters'**. Inside the **main()** method, d1 is an object of a Distance class.

Output



Friend function can also be useful when we are working on objects of two different classes.

Let's understand through an example.

```
1. // Add members of two different classes using friend functions
2. #include <iostream>
3. using namespace std;
4. // forward declaration of a class
5. class ClassB;
6. // declaration of a class
7. class ClassA {
8.  public:
9.         // constructor ClassA() to initialize num1 to 12
10.        ClassA()
11.        {
12.            num1 =12;
```

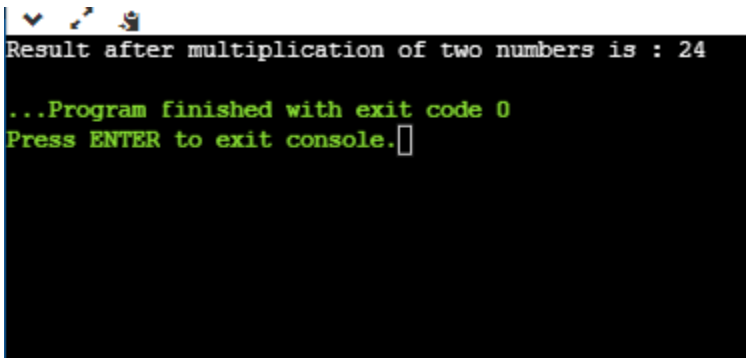
```
13.       }
14.    private:
15.       int num1;  // declaration of integer variable
16.        // friend function declaration
17.        friend int multiply(ClassA, ClassB);
18. };
19. class ClassB {
20. public:
21.       // constructor ClassB() to initialize num2 to 2
22.       ClassB()
23.       {
24.          num2 = 2;
25.       }
26.    private:
27.       int num2;  // declaration of integer variable
28.  // friend function declaration
29.       friend int multiply(ClassA, ClassB);
30. };
31.
32. // access members of both classes
33. int multiply(ClassA object1, ClassB object2)
34. {
35.    return (object1.num1 * object2.num2);
36. }
37.
38. int main() {
39.    ClassA object1;   // declaration of object of ClassA
40.    ClassB object2;   // declaration of object of ClassB
```

```
41.    cout << "Result after multiplication of two numbers is : " << multiply(object1,
       object2);
42.    return 0;
43.}
```

In the above code, we have defined two classes named as **ClassA** and **ClassB**. Both these classes contain the friend function '**multiply()**'. The friend function can access the data members of both the classes, i.e., **ClassA** and **ClassB**. The **multiply()** function accesses the **num1** and **num2** of **ClassA** and **ClassB** respectively. In the above code, we have created object1 and object2 of ClassA and ClassB respectively. The multiply() function multiplies the num1 and num2 and returns the result.

As we can observe in the above code that the friend function in **ClassA** is also using **ClassB** without prior declaration of **ClassB**. So, in this case, we need to provide the forward declaration of **ClassB**.

Output



Friend class in C++

We can also create a friend class with the help of **friend** keyword.

```
1. class Class1;
2. class Class2
3. {
4.   // Class1 is a friend class of Class2
5.     friend class Class1;
```

```
    .. .....
}
class Class1
{
 ....
}
```

In the above declaration, Class1 is declared as a friend class of Class2. All the members of Class2 can be accessed in Class1.

Let's understand through an example.


```
// C++ program to demonstrate the working of friend class
#include <iostream>
using namespace std;
// forward declaration
class ClassB;

class ClassA {
  private:
    int num1;

    // friend class declaration
    friend class ClassB;


  public:
    // constructor to initialize numA to 10
    ClassA()
```

```
    {
        num1 = 10;
    }
};
class ClassB {
  private:
    int num2;
  public:
    // constructor to initialize numB to 1
    ClassB()
    {
        num2 = 1;
    }
  // member function to add num1
  // from ClassA and num2 from ClassB
  int add() {
    ClassA objectA;
    return objectA.num1 + num2;
  }
};
int main() {
  ClassB objectB;
  cout << "Sum: " << objectB.add();
  return 0;
}
```

In the above code, we have created two classes named as ClassA and ClassB. Since ClassA is declared as friend of ClassB, so ClassA can access all the data members of ClassB. In ClassB, we have defined a function add() that returns the sum of num1 and

num2. Since ClassB is declared as friend of ClassA, so we can create the objects of ClassA in ClassB.

Output