

**SSBT's College of Arts, Commerce & Science, Bambhori, Jalgaon**  
**Bachelor Of Computer Application (B.C.A)**

**Experiment: 01**

**DOP:**

**DOC:**

---

**Aim:** Demonstrate the use of if...else and control structure (even or odd, sum & average, prime or composite, sum\_difference\_product\_quotient of two integers).

**Theory:** The if-else statement in C is a **flow control statement** used for **decision-making** in the C program. It is one of the core concepts of C programming. It is an extension of the **if in C** that includes an **else** block along with the already existing if block.

### **C if Statement**

The if statement in C is used to execute a block of code based on a specified condition.

The syntax of the **if statement** in C is:

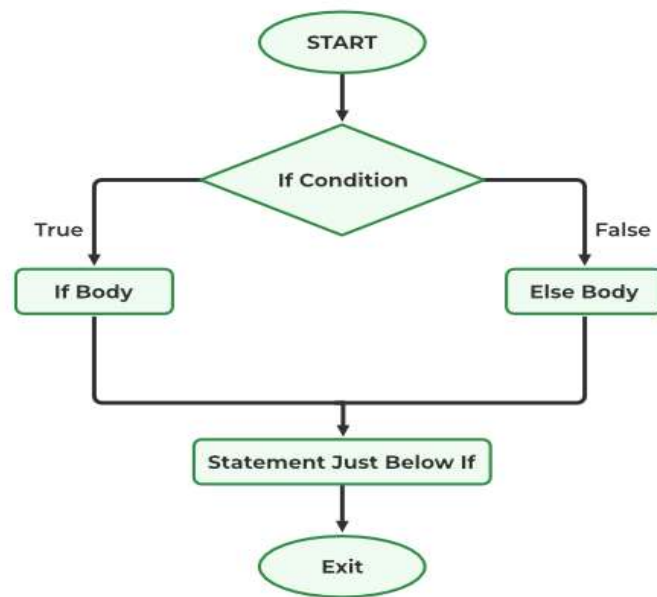
```
if (condition) {  
    // code to be executed if the condition is true  
}
```

### **C if-else Statement**

The if-else statement is a decision-making statement that is used to decide whether the part of the code will be executed or not based on the **specified condition (test expression)**. If the given condition is true, then the code inside the if block is executed, otherwise the code inside the else block is executed.

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

## Flowchart of the if-else statement



### 1.1] Even or odd.

#### A] Code :-

```
#include <iostream>
using namespace std;
int main(){
    int num;
    cout<<"\nEnter no to check even or odd : ";
    cin>>num;
    if(num%2==0){
        cout<<num<<" is an even number."<<endl;
    }
    else{
        cout<<num<<" is an odd number."<<endl;
    }
    return 0;
}
```

#### B] O/P :-

##### B.1] :-

```
Enter no to check even or odd : 4
4 is an even number.
```

##### B.2] :-

```
Enter no to check even or odd : 5
5 is an odd number.
```

## 1.2] Sum & Average.

### A] Code :-

```
#include <iostream>
using namespace std;
int main()
{
    int i, sum = 0, avg;
    for(i = 1; i <= 10; i++)
    {
        sum += i;
        cout<<i<<" ";
    }
    avg=sum / --i;

    cout<<"\nSum = "<<sum;

    cout<<"\nAverage = "<<avg;

}
```

### B] O/P :-

```
12345678910
Sum = 55
Average = 5.5
```

## 1.3] Prime or Composite:

```
#include <iostream>
using namespace std;
int main()
{
    int n, i, m = 0, flag = 0;
    cout<<"Enter a number to check if its Prime : ";
    cin>>n;
    m = n / 2;
    for(i = 2; i <= m; i++)
    {
        if(n % i == 0)
        {
            cout<<n<<" is not a Prime Number."<<endl;
            flag = 1;
            break;
        }
    }
    if(flag == 0)
    {
        cout<<n<<" is a Prime number."<<endl;
    }
    return 0;
}
```

### B] O/P :-

**B.1] :-**

```
Enter a number to check if its Prime : 4
4 is not a Prime Number.
```

**B.2] :-**

```
Enter a number to check if its Prime : 5
5 is a Prime number.
```

#### **1.4] Sum, Difference, Product & Quotient.**

**A] Code :-**

```
#include <iostream>
using namespace std;
int main()
{
    int a, b;
    cout<<"\nEnter the value of A and B : ";
    cin>>a>>b;
    // Sum (+)
    cout<<a<<" + "<<b<<" = "<<a+b<<endl;
    // Subtraction (-)
    cout<<a<<" - "<<b<<" = "<<a-b<<endl;
    // Product (*)
    cout<<a<<" * "<<b<<" = "<<a*b<<endl;
    // Division/Quotient (/)
    cout<<a<<" / "<<b<<" = "<<a/b<<endl;
    return 0;
}
```

**B] O/P :-**

```
Enter the value of A and B : 6 5
6 + 5 = 11
6 - 5 = 1
6 * 5 = 30
6 / 5 = 1
```

**Conclusion:**\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

**Submitted By :**

**Name:**

**Sign:**

**Roll No :**

**Checked By :**

**Name: Ms. Pratiksha S. Patil.**

**Sign:**

**SSBT's College of Arts, Commerce & Science, Bambhori, Jalgaon**  
**Bachelor Of Computer Application (B.C.A)**

**Experiment: 02**

**DOP:**

**DOC:**

---

**Aim:** Write a program to demonstrate use of Function overloading.

**Theory:** More than one function of the same name in the same scope. These functions are called *overloaded* functions, or *overloads*. Overloaded functions enable you to supply different semantics for a function, depending on the types and number of its arguments.

Two or more functions may share the same name but not the same list of arguments when functional overloading occurs. It is a key characteristic of C++. Compile-time polymorphism and function overloading are similar concepts. A close examination reveals that the name stays the same, although the list of arguments, data type, and order all change. Take a look at a C++ function overloading example.

The **advantage** of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

```
#include<iostream>
using namespace std;
class Cal
{
    public:
        int add(int a,int b)
        {
            return a + b;
        }
        int add(int a, int b, int c)
        {
            return a + b + c;
        }
};
int main(void)
{
    Cal C;                                // class object declaration.
    cout<<C.add(10, 20)<<endl;
    cout<<C.add(12, 20, 23);
    return 0;
}
```

**Output:**

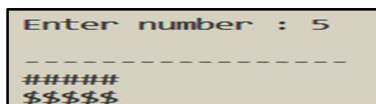
**30**

**35**

**A] Code :-**

```
#include <iostream>
using namespace std;
void print()
{
    cout<<"\n-----";
}
void print(int a)
{
    cout<<"\n";
    for(int i = 1; i <= a; i++)
    {
        cout<<"#";
    }
}
void print(int a, char c)
{
    cout<<"\n";
    for(int i = 1; i <= a; i++)
    {
        cout<<c;
    }
}
int main()
{
    int x;
    cout<<"\nEnter number : ";
    cin>>x;
    print();
    print(x);
    print(x, '$');
    return 0;
}
```

**B] O/P :-**



```
Enter number : 5
-----
#####
$$$$$
```

**Conclusion:** \_\_\_\_\_

\_\_\_\_\_

**Submitted By :**

**Name:**

**Sign:**

**Roll No :**

**Checked By :**

**Name: Ms. Pratiksha S. Patil.**

**Sign:**

**SSBT's College of Arts, Commerce & Science, Bambhori, Jalgaon**  
**Bachelor Of Computer Application (B.C.A)**

**Experiment: 03**

**DOP:**

**DOC:**

**Aim:** Write a program to demonstrate encapsulation using class.

**Theory:** Encapsulation is a way to restrict the direct access to some components of an object, so users cannot access state values for all of the variables of a particular object. Encapsulation can be used to hide both data members and data functions or methods associated with an instantiated class or object.

**Benefits of encapsulation programming**

Encapsulation in programming has a few key benefits.

These include:

- **Hiding data:** Users will have no idea how classes are being implemented or stored. All that users will know is that values are being passed and initialized.
- **More flexibility:** Enables you to set variables as read or write-only. Examples include: setName(), setAge() or to set variables as write-only then you only need to omit the get methods like getName(), getAge() etc.
- **Easy to reuse:** With encapsulation, it's easy to change and adapt to new requirements

"Abstraction and encapsulation are complementary concepts: abstraction focuses on the observable behavior of an object...encapsulation focuses on the implementation that gives rise to this behavior"

Abstraction occurs when a programmer hides any irrelevant data about an object or an instantiated class to reduce complexity and help users interact with a program more efficiently.

The term abstraction vs encapsulation can be used to describe the process of hiding some of the information contained in an object or class, but it may also refer to the object itself.

An abstraction is any named entity that contains a selection of data and behaviors specific to a particular usage of the originating entity.



### A]Code:-

```
#include <iostream>
using namespace std;
class Addition
{
    private:
        int a, b;
    public:
        void getNum()
        {
            cout<<"Enter value of a & b : ";
            cin>>a>>b;
        }
        void printSum()
        {
            cout<<a<<" + "<<b<<" = "<<a+b<<endl;
        }
};
int main()
{
    Addition N;
    N.getNum();
    N.printSum();
    return 0;
}
```

### B] O/P :-

```
Enter value of a & b : 6 5
6 + 5 = 11
```

Conclusion: \_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

**Submitted By :**

**Name:**

**Sign:**

**Roll No :**

**Checked By :**

**Name: Ms. Pratiksha S. Patil.**

**Sign:**

**SSBT's College of Arts, Commerce & Science, Bambhori, Jalgaon**  
**Bachelor Of Computer Application (B.C.A)**

**Experiment: 04**

**DOP:**

**DOC:**

**Aim:** Write a program to demonstrate use of Constructors and Destructor.

**Theory:** Constructor and Destructor are the special member functions of the class which are created by the C++ compiler or can be defined by the user.

The constructor is used to initialize the object of the class while the destructor is called by the compiler when the object is destroyed.

While programming, sometimes there might be the need to initialize data members and member functions of the objects before performing any operations. Data members are the variables declared in any class by using fundamental data types (like int, char, float, etc.) or derived data types (like class, structure, pointer, etc.). The functions defined inside the class definition are known as member functions.

Suppose you are developing a game. In that game, each time a new player registers, we need to assign their initial location, health, acceleration, and certain other quantities to some default value.

This can be done by defining separate functions for each quantity and assigning the quantities to the required default values. For this, we need to call a list of functions every time a new player registers. Now, this process can become lengthy and complicated.

```
class scaler
{
    public:
    //Constructor
    scaler()
    {
        //Constructor Body;
    }
}
```

### A] Code :-

```
#include <iostream>
using namespace std;

class MyClass {
public:
    // Constructor
    MyClass() {
        cout << "Constructor called" << endl;
    }

    // Destructor
    ~MyClass() {
        cout << "Destructor called" << endl;
    }
};

int main() {
    cout << "Creating an object..." << endl;
    MyClass obj; // Constructor is called when the
    object is created

    cout << "Doing some work with the object..." <<
    endl;

    // Destructor is called when the object goes out
    of scope (end of the block)

    cout << "Object is about to go out of scope..."
    << endl;

    return 0; // Here, the destructor will be called
    for obj
}
```

### B] O/P :-

```
Creating an object...
Constructor called
Doing some work with the object...
Object is about to go out of scope...
Destructor called
```

Conclusion: \_\_\_\_\_

Submitted By :

Name:

Sign:

Roll No :

Checked By :

Name: Ms. Pratiksha S. Patil.

Sign:

**SSBT's College of Arts, Commerce & Science, Bambhori, Jalgaon**  
**Bachelor Of Computer Application (B.C.A)**

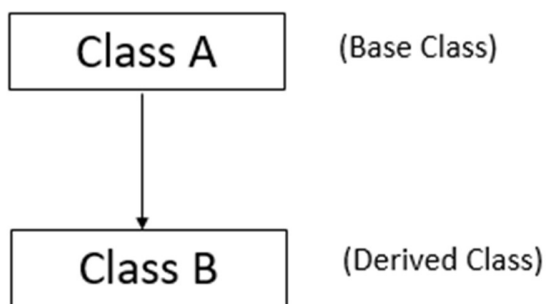
**Experiment: 5**

**DOP:**

**DOC:**

**Aim:** Write a program to demonstrate Single Inheritance.

**Theory:** Single inheritance is one type of inheritance in which the derived class inherits only one base class. It provides reusability by allowing the derived class to inherit the features of the base class using objects. A class whose properties are inherited for reusability is called parent class or superclass or base class. The class that inherits the properties from this base or superclass is called a child class or derived class or subclass. Let's learn further about the concept of single inheritance in C++.



**Syntax of Single Inheritance in C++:**

All the properties and methods can be inherited from the base class to the derived class.

```
class base_class
{
//code
};
class derived_class : public(access_modifier) base_class
{
//code
};
int main()
{
base_class object_1;
derived_class object_2;
//code
}
```

## Examples of Single Inheritance in C++

Let us find different examples of single inheritance of C++ below:

### ***Example #1***

```
#include <iostream>
using namespace std;
class First
{
public :void display()
{
cout<<"This display is inside the first class" << endl;
}
};
class Second: public First
{
public: void show()
{
cout<<"This show is inside the second class which is derived
from parent class" << endl;
}
};
int main()
{
First f;
f.display();
//f.show();
Second s;
s.display();
s.show();
}
```

If we uncomment the f.show() method, we will get the below error.

**Code Explanation:** Now, let us see how the code is actually functioning. We have created two classes, namely First and Second. We have derived the second class from the first. We have one function under the base class and another function in the derived class. In the main function, we have declared our objects for both the parent and the child class. With the child class object, we tried to access both the base and derived class methods, which would be absolutely successful.

But if we try to access the derived class method or variables through the base class object as seen in the second output, we get the error. It is obvious because the base class object cannot access the derived class methods/ variables, but vice versa.

## ***Example #2***

Now, we can check the output by giving the same methods in both classes. The code is written below.

```
#include <iostream>
using namespace std;
class First
{
public :void display()
{
cout<<"This display is inside the first class" << endl;
}
};
class Second: public First
{
public: void display()
{
cout<<"This show is inside the second class which is derived
from parent class" << endl;
}
```

```
};
int main()
{
    First f;
    f.display();
    f.display();
    Second s;
    s.display();
    s.display();
}
```

**Code Explanation:** It is the same code, except for the method name in the derived class is now the same as the method name in the base class. All the methods would give the same output. With the concept of overriding, the classes and their respective objects would find their own method name and display the contents of the same class.

Let us see the working of single inheritance in C++ with the help of the examples below.

### ***Example:***

```
#include <iostream>
using namespace std;
class Sum_and_mul
{
public:
    int c=10;
    public :
    void sum_1(int a, int b)
    {
        int result;
        result=a+c;
        cout<<" The result for sum of a and c is: "<<result<<endl;
    }
    void mul_1(int a,int b)
    {
        int result;
        result=a*c;
```

```
cout<<" The result for multiplication of a and c is:
"<<result<<endl;
}
};
class Mul_and_sum : public Sum_and_mul
{
int d=20;
public:
void sum_2()
{
int result;
result=c+d;
cout<<" The result for sum of c and d is: "<<result<<endl;
}
void mul_2()
{
int result;
result=c*d;
cout<<" The result for multiplication of c and d is:
"<<result<<endl;
}
};
int main()
{
int a,b;
cout<<" Enter value for a: ";
cin>>a;
cout<<" Enter value for b: ";
cin>>b;
Sum_and_mul sam;
Mul_and_sum mas;
sam.sum_1(a,b);
sam.mul_1(a,b);
mas.sum_1(a,b);
mas.mul_1(a,b);
mas.sum_2();
mas.mul_2();
}
```



**Program Explanation:** In the above example that is given,

- We had two classes, Sum\_and\_mul and Mul\_and\_sum, as a base and derived classes, respectively.
- There are two methods and a single variable pertaining to those two classes.
- We then declared those variables and methods for each class.
- We had inherited the properties of base class to derived class by using 'colon (:)'
- Here, it can be observed that the derived class methods have the variables of the base class in performing a few mathematical operations.
- The vice versa in using the derived class variable in the base class is not possible. Give it a try if you want to check out the error.
- Then we had created our objects for each class.
- With the created object for the derived class, we had handled both base class and derived class methods, and we have got the output perfectly.

## ***Example #2***

Let us see how we can handle methods between the base and derived classes in the below example.

**Code:**

```
#include <iostream>
using namespace std;
class AB
{
int a = 10;
int b = 20;
public:
```

```
int sub()
{
int r = b-a;
return r;
}
};

class BA : public AB
{
public:
void show()
{
int s = sub();
cout <<"Subtraction of b and a is : "<<s<< endl;
}
};

int main()
{
BA b;
b.show();
return 0;
}
```

Conclusion: \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

**Submitted By :**

**Name:**

**Sign:**

**Roll No :**

**Checked By :**

**Name: Ms. Pratiksha S. Patil.**

**Sign:**

**SSBT's College of Arts, Commerce & Science, Bambhori, Jalgaon**  
**Bachelor Of Computer Application (B.C.A)**  
**Experiment: 06**

**DOP:**

**DOC:**

**Aim:** Write a program to demonstrate multiple inheritances.

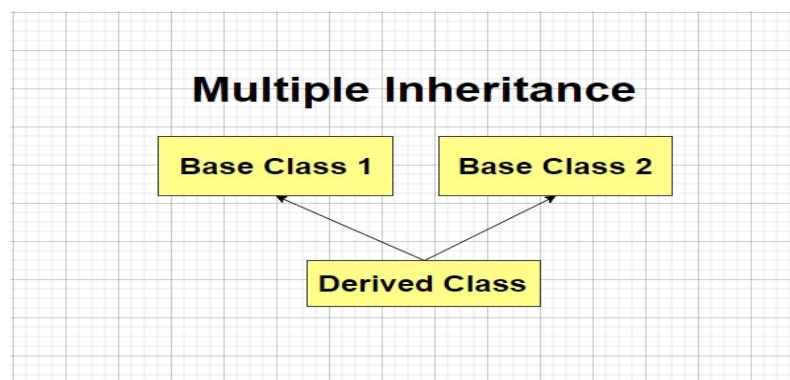
**Theory:** This section will discuss the Multiple Inheritances in the C++ programming language. When we acquire the features and functionalities of one class to another class, the process is called **Inheritance**.

In this way, we can reuse, extend or modify all the attributes and behaviour of the parent class using the derived class object. It is the most important feature of object-oriented programming that reduces the length of the program.

A class that inherits all member functions and functionality from another or parent class is called the **derived class**. And the class from which derive class acquires some features is called the **base or parent class**.

**Multiple Inheritance** is the concept of the Inheritance in C++ that allows a child class to inherit properties or behaviour from multiple **base** classes. Therefore, we can say it is the process that enables a derived class to acquire member functions, properties, characteristics from more than one base class.

Following is the diagram of the Multiple Inheritances in the C++ programming language.



In the above diagram, there are two-parent classes: **Base Class 1** and **Base Class 2**, whereas there is only one **Child Class**. The Child Class acquires all features from both Base class 1 and Base class 2. Therefore, we termed the type of Inheritance as Multiple Inheritance.

## Syntax of the Multiple Inheritance

```
class A
{
// code of class A
}
class B
{
// code of class B
}
class C: public A, public B (access modifier class_name)
{
// code of the derived class
}
```

In the above syntax, class A and class B are two base classes, and class C is the child class that inherits some features of the parent classes.

Let's write the various program of Multiple Inheritance to inherit the member functions and functionality from more than one base class using the derived class.

### Example 1: Program to use the Multiple Inheritance:

#### **Program1.cpp**

```
#include <iostream>
using namespace std;

// create a base class1
class Base_class
{
    // access specifier
    public:
    // It is a member function
    void display()
    {
        cout << " It is the first function of the Base class " << endl;
    }
};

// create a base class2
class Base_class2
```

```

{
    // access specifier
    public:
    // It is a member function
    void display2()
    {
        cout << " It is the second function of the Base class " << endl;
    }
};

/* create a child_class to inherit features of Base_class and Base_class2 with access specifier. */
class child_class: public Base_class, public Base_class2
{

    // access specifier
    public:
    void display3() // It is a member function of derive class
    {
        cout << " It is the function of the derived class " << endl;
    }

};

int main ()
{
    // create an object for derived class
    child_class ch;
    ch.display(); // call member function of Base_class1
    ch.display2(); // call member function of Base_class2
    ch.display3(); // call member function of child_class
}

```

### Output

```

It is the first function of the Base class
It is the second function of the Base class
It is the function of the derived class

```

### A] Code :-

```
#include <iostream>
using namespace std;
class A
{
    public:
        int x;
        void getx()
        {
            cout<<"Enter the value of x : ";
            cin>>x;
        }
};
class B
{
    public:
        int y;
        void gety()
        {
            cout<<"Enter the value of y : ";
            cin>>y;
        }
};
class C : public A, public B
{
    public:
        void sum()
        {
            cout<<x<<" + "<<y<<" = "<<x+y;
        }
};
int main()
{
    C obj;
    obj.getx();
    obj.gety();
    obj.sum();

    return 0;
}
```

### B] O/P :-

```
Enter the value of x : 5
Enter the value of y : 6
5 + 6 = 11
```

Conclusion: \_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

**Submitted By :**

**Name:**

**Sign:**

**Roll No :**

**Checked By :**

**Name: Ms. Pratiksha S. Patil.**

**Sign:**

**SSBT's College of Arts, Commerce & Science, Bambhori, Jalgaon**  
**Bachelor Of Computer Application (B.C.A)**  
**Experiment: 07**

**DOP:**

**DOC:**

**Aim:** Write a program to demonstrate use of Operator overloading using friend function.

**Theory:** In this article, I am going to discuss **Operator Overloading using Friend function in C++** with Examples. Please read our previous article where we discussed **Operator Overloading in C++** with Examples. C++ Programming Language provides a special mechanism to change the current functionality of some operators within its class which is often called operator overloading. Operator Overloading is the method by which we can change the function of some specific operators to do some different tasks.

### Friend Function Operator Overloading:

In our previous article, we have already seen how to overload unary (++ , -) and binary (+) operators in C++ with Examples. There is one more method for overloading an operator in C++ that is using the friend function. Let us learn it through the same example that is using the same Complex class. The following is the sample code that we have created in our previous article.

Suppose we want to add two complex numbers i.e. C1 and C2,

**C3 = C1 + C2;**

We have created add function in class Complex. At that time the idea was that either C1 will add to C2 or C2 will add to C1. But now we want somebody else to add two complex numbers.

We have also given you an example that if 'X' has some money and 'Y' also has some money and they wanted to add their money. So 'X' can add money or 'Y' can add money or they can also take help from another person i.e. friends. If their friend is adding the money then they both have to give their money to him as the parameter. Then only their friend can add the money. So, the same approach will follow for the friend function. Here we are writing the Complex class,

In the above example, we have created two integer type private data members real and img. Then we overloaded the + operator with two parameters C1 and C2. We have not defined the body here. We have made it a friend by using the friend function. This is the prototype of the friend function in C++. This function will return an object of type Complex.

So, this friend function will take two complex numbers as parameters and return a Complex number.

**C3 = C1 + C2;**

It is just like there is a function that will take C1 and C2 as parameters and add them and return the result. So, neither C1 nor C2 adding but someone else is adding. This friend function has to be written outside the class without using scope resolution. Now let us write the body of the friend function 'operator +' outside the class,

This function doesn't belong to the class but it is a friend of the Complex class. So, we don't use any scope resolution operator. So, this is another approach to overloading operators in

C++. So, operators, we can overload as a member function as well as we can overload them as friend functions. Now let us write the complete program in C++.

### Points to Remember While Overloading Operator using Friend Function:

We need to remember the following pointers while working with Operator Overloading in C++ Using Friend Function.

1. The Friend function in C++ using operator overloading offers better flexibility to the class.
2. The Friend functions are not a member of the class and hence they do not have 'this' pointer.
3. When we overload a unary operator, we need to pass one argument.
4. When we overload a binary operator, we need to pass two arguments.
5. The friend function in C++ can access the private data members of a class directly.
6. An overloaded operator friend could be declared in either the private or public section of a class.
7. When redefining the meaning of an operator by operator overloading the friend function, we cannot change its basic meaning. For example, we cannot redefine minus operator + to multiply two operands of a user-defined data type.

### Using Friend Function to Overload Unary Operator:

We can also overload a unary operator in C++ by using a friend function. The overloaded ++ operator relative to the Test class using a member function is shown in the below example.

**Note:** In general, you should define the member function to implement operator overloading, friend function has been introduced for a different purpose that we are going to discuss in our upcoming articles.

In the next article, I am going to discuss **Insertion Operator Overloading in C++** with Examples. Here, in this article, I try to explain **Operator Overloading using Friend Function in C++** with Examples and I hope you enjoy this Operator Overloading using Friend Function in C++ with Examples article. I would like to have your feedback. Please post your feedback, question, or comments about this article.

```
class Complex
{
    private:
        int real;
        int img;
    public:
        Complex (int r = 0, int i = 0)
        {
            real = r;
            img = i;
        }
        Complex add (Complex x)
        {
            Complex temp;
            temp.real = real + x.real;
            temp.img = img + x.img;
            return temp;
        }
}
```



```
        void Display()
        {
            cout << real << "+i" << img << endl;
        }
};

int main()
{
    Complex C1 (3, 7);
    C1.Display();
    Complex C2 (5, 2);
    C2.Display();
    Complex C3;
    C3 = C1.add (C2); // C2.add(C1);
    C3.Display();
}
```

**Conclusion:**\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

**Submitted By :**

**Name:**

**Sign:**

**Roll No :**

**Checked By :**

**Name: Ms. Pratiksha S. Patil.**

**Sign:**

**SSBT's College of Arts, Commerce & Science, Bambhori, Jalgaon**  
**Bachelor Of Computer Application (B.C.A)**

**Experiment: 08**

**DOP:**

**DOC:**

---

**Aim:** Write a program to demonstrate use of friend function.

**Theory:** A friend function in C++ is defined as a function that can access private, protected, and public members of a class.

The friend function is declared using the friend keyword inside the body of the class.

**Friend Function Syntax:**

```
class className
{
    ... ..
    friend returnType functionName(arguments);
    ... ..
}
```

By using the keyword, the '**friend**' compiler understands that the given function is a friend function.

We declare a friend function inside the body of a class, whose private and protective data needs to be accessed, starting with the keyword friend to access the data. We use them when we need to operate between two different classes at the same time.

**What is Friend Function?**

**Friend functions** of the class are granted permission to access private and protected members of the class in C++. They are defined globally outside the class scope. Friend functions are not member functions of the class. So, what exactly is the friend function?

A friend function in C++ is a function that is declared outside a class but is capable of accessing the private and protected members of the class. There could be situations in programming wherein we want two classes to share their members. These members may be data members, class functions or function templates. In such cases, we make the desired function, a friend to both these classes which will allow accessing private and protected data of members of the class.

Generally, non-member functions cannot access the private members of a particular class. Once declared as a friend function, the function is able to access the private and protected members of these classes.

Friend functions in C++ have the following types

- Function with no argument and no return value
- Function with no argument but with return value
- Function with argument but no return value
- Function with argument and return value
- we require friend functions whenever we have to access the private or protected members of a class. This is only the case when we do not want to use the objects of that class to access these private or protected members.

To understand this better, let us consider two classes: Tokyo and Rio.

We might require a function, metro(), to access both these classes without any restrictions. Without the friend function, we will require the object of these classes to access all the members.

Friend functions in c++ help us avoid the scenario where the function has to be a member of either of these classes for access.

#### **Characteristics of Friend Function in C++:**

- The function is not in the 'scope' of the class to which it has been declared a friend.
- Friend functionality is not restricted to only one class
- Friend functions can be a member of a class or a function that is declared outside the scope of class.
- It cannot be invoked using the object as it is not in the scope of that class.
- We can invoke it like any normal function of the class.
- Friend functions have objects as arguments.
- It cannot access the member names directly and has to use dot membership operator and use an object name with the member name.
- We can declare it either in the 'public' or the 'private' part.
- These are some of the friend functions in C++ characteristics

```

#include <iostream>
using namespace std;
class MyClass; // Forward declaration

class FriendClass
{
    public:
        // Friend function declaration
        friend void friendFunction(const MyClass& obj);
};
class MyClass
{
    private:
        int data;
    public:
        MyClass(int value) : data(value) {}
        // Friend function definition
        friend void friendFunction(const MyClass& obj);
        void display()
        {
            cout << "Data : " << data << endl;
        }
};
// Friend function definition
void friendFunction(const MyClass& obj)
{
    cout << "Friend function can access MyClass's private data : " << obj.data
<< endl;
}
int main()
{
    MyClass obj(42);
    cout << "Calling the friend function from main..." << endl;
    friendFunction(obj); // Call the friend function
    cout << "Calling the member function from main..." << endl;
    obj.display(); // Call the member function
    return 0;
}

```

**O/P :-**

```

Calling the friend function from main...
Friend function can access MyClass's private data : 42
Calling the member function from main...
Data : 42

```

**Conclusion:** \_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

**Submitted By :**

**Name:**

**Sign:**

**Roll No :**

**Checked By :**

**Name: Ms. Pratiksha S. Patil.**

**Sign:**

**SSBT's College of Arts, Commerce & Science, Bambhori, Jalgaon**  
**Bachelor Of Computer Application (B.C.A)**

**Experiment: 09**

**DOP:**

**DOC:**

---

**Aim:** Write a program to demonstrate use of Virtual functions.

**Theory:** A virtual function in C++ is a base class member function that you can redefine in a derived class to achieve polymorphism. You can declare the function in the base class using the virtual keyword. Once you declare the function in the base class, you can use a pointer or reference to call the virtual class and execute its virtual version in the derived class. Thus, it asks the compiler to determine the object's type during run-time and create a function bind (late binding or dynamic linkage).

A virtual function in C++ helps ensure you call the correct function via a reference or pointer. The C++ programming language allows you only to use a single pointer to refer to all the derived class objects. Since the pointer refers to all the derived objects, calling it will consistently execute the function in the base class. You can overcome this challenge with a virtual function in C++ as it helps execute the virtual version of the derived class, which is done at the run-time.

**Rules of Virtual Function:**

There are a few rules you need to follow to create a virtual function in C++. These rules are:

- The functions cannot be static
- You derive them using the "virtual" keyword
- Virtual functions in C++ needs to be a member of some other class (base class)
- They can be a friend function of another class
- The prototype of these functions should be the same for both the base and derived class
- Virtual functions are accessible using object pointers
- Redefining the virtual function in the derived class is optional, but it needs to be defined in the base class
- The function call resolving is done at run-time
- You can create a virtual destructor but not a constructor

```

#include <iostream>
using namespace std;
class Base
{
    public:
    virtual void Output()
    {
        cout << "Output Base class" << endl;
    }
    void Display()
    {
        cout << "Display Base class" << endl;
    }
};
class Derived : public Base
{
    public:
    void Output()
    {
        cout << "Output Derived class" << endl;
    }
    void Display()
    {
        cout << "Display Derived class" << endl;
    }
};
int main()
{
    Base* bptr;
    Derived dptr;
    bptr = &dptr;
    // virtual function binding
    bptr->Output();
    // Non-virtual function binding
    bptr->Display();
}

```

*Output:*

```

Output Derived Class
Display Base Class

```

### **Pure Virtual Function:**

A pure virtual function in C++, also known as the do-nothing function, is a virtual function that does not perform any task.

It is only used as a placeholder and does not contain any function definition (do-nothing function). It is declared in an abstract base class. These types of classes cannot declare any objects of their own. You derive a pure virtual function in C++ using the following syntax:

```
Virtual void class_name() = 0;
```

### Example of Pure Virtual Functions in C++

```
#include <iostream>
using namespace std;
class Base
{
    public:
        virtual void Output() = 0;
};
class Derived : public Base
{
    public:
        void Output()
        {
            std::cout << "Class derived from the Base class." << std::endl;
        }
};
int main()
{
    Base *bptr;
    Derived dptr;
    bptr = &dptr;
    bptr->Output();
    return 0;
}
```

#### *Output:*

Class derived from the Base class.

**Conclusion:** \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

**Submitted By :**

**Name:**

**Sign:**

**Roll No :**

**Checked By :**

**Name: Ms. Pratiksha S. Patil.**

**Sign:**

**SSBT's College of Arts, Commerce & Science, Bambhori, Jalgaon**  
**Bachelor Of Computer Application (B.C.A)**

**Experiment: 10**

**DOP:**

**DOC**

---

**Aim:** Write a program to demonstrate use of pointer : -

- a) Write a program to demonstrate use of pointer to pointer.
- b) Write a program to demonstrate use of pointer to objects.
- c) Write a program to demonstrate use of pointer to function.”

**Theory:** A pointer is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it.

The general form of a pointer variable declaration is –

```
type *var-name;
```

Here, type is the pointer's base type; it must be a valid C++ type and var-name is the name of the pointer variable.

The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer.

Following are the valid pointer declaration –

```
int *ip; // pointer to an integer
double *dp; // pointer to a double
float *fp; // pointer to a float
char *ch // pointer to character
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address.

The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.



**Pointer Arithmetic :-** As you understood pointer is an address which is a numeric value; therefore, you can perform arithmetic operations on a pointer just as you can a numeric value.

There are four arithmetic operators that can be used on pointers: ++, --, +, and -.

To understand pointer arithmetic, let us consider that **ptr** is an integer pointer which points to the address 1000.

Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer –

`ptr++`

the **ptr** will point to the location 1004 because each time ptr is incremented, it will point to the next integer.

This operation will move the pointer to next memory location without impacting actual value at the memory location.

If ptr points to a character whose address is 1000, then above operation will point to the location 1001 because next character will be available at 1001.

```
#include <iostream>
using namespace std;

int main() {
    int number = 42;
    double pi = 3.14159;
    char letter = 'A';

    // Declare and initialize pointers
    int* intPtr = &number;
    double* doublePtr = &pi;
    char* charPtr = &letter;

    // Display the values through pointers
    cout << "Value of number using intPtr: " << *intPtr << endl;
    cout << "Value of pi using doublePtr: " << *doublePtr << endl;
    cout << "Value of letter using charPtr: " << *charPtr << endl;

    // Modify values through pointers
    *intPtr = 10;
    *doublePtr = 2.71828;
    *charPtr = 'B';

    // Display the modified values
    cout << "Modified value of number: " << number << endl;
    cout << "Modified value of pi: " << pi << endl;
    cout << "Modified value of letter: " << letter << endl;

    return 0;
}
```

**O/P: -**

```
Value of number using intPtr: 42
Value of pi using doublePtr: 3.14159
Value of letter using charPtr: A
Modified value of number: 10
Modified value of pi: 2.71828
Modified value of letter: B
```

**a] :-**

```
#include <iostream>

using namespace std;

int main() {

    int number = 42;

    int* ptr = &number; // Pointer to an integer

    int** ptrToPtr = &ptr; // Pointer to a pointer to an integer
    (double pointer)

    // Display the values

    cout << "Value of number: " << number << endl;

    cout << "Value of number using ptr: " << *ptr << endl;

    cout << "Value of number using ptrToPtr: " << **ptrToPtr << endl;

    // Modify the value through ptrToPtr

    **ptrToPtr = 99;

    // Display the modified value

    cout << "Modified value of number: " << number << endl;

    return 0;
}
```

**O/P :-**

```
Value of number: 42
Value of number using ptr: 42
Value of number using ptrToPtr: 42
Modified value of number: 99
```

**b] :-**

```
#include <iostream>
using namespace std;

class MyClass
{
    public:
        int data;

        MyClass(int value) : data(value) {}

        void display(){
            cout << "Data : " << data << endl;
        }
};

int main(){
    //Create an object of MyClass
    MyClass obj(42);

    //Create a pointer to an object of MyClass
    MyClass* ptr = &obj;

    //Access and modify object's members through pointer
    ptr->display(); //Access member function
    cout << "Value of data using pointer : " << ptr->data << endl;

    //Modify object's member through the pointer
    ptr->data = 99;
    cout << "Modified value of data : " << obj.data << endl;

    return 0;
}
```

**O/P :-**

```
Data : 42
Value of data using pointer : 42
Modified value of data : 99
```

**c] :-**

```
#include <iostream>
using namespace std;

// Function that adds two numbers
int add(int a, int b) {
    return a + b;
}

// Function that subtracts two numbers
int subtract(int a, int b) {
    return a - b;
}

int main() {
    // Declare a pointer to a function that takes two integers
    and returns an integer
    int (*operation)(int, int);

    // Assign the pointer to the add function
    operation = add;

    // Use the pointer to call the add function
    int result = operation(10, 5);
    cout << "Result of addition: " << result << endl;

    // Reassign the pointer to the subtract function
    operation = subtract;

    // Use the pointer to call the subtract function
    result = operation(10, 5);

    cout << "Result of subtraction: " << result << endl;

    return 0;
}
```

**B] O/P :-**

```
Result of addition: 15
Result of subtraction: 5
```

**Conclusion:** \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

**Submitted By :**

**Name:**

**Sign:**

**Roll No :**

**Checked By :**

**Name: Ms. Pratiksha S. Patil.**

**Sign:**

**SSBT's College of Arts, Commerce & Science, Bambhori, Jalgaon**  
**Bachelor Of Computer Application (B.C.A)**

**Experiment: 11**

**DOP:**

**DOC:**

---

**Aim:** Write a program to demonstrate use of Exception Handling.

**Theory:** When executing C++ code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, C++ will normally stop and generate an error message. The technical term for this is: C++ will throw an **exception** (throw an error).

**try and catch:**

Exception handling in C++ consist of three keywords: `try`, `throw` and `catch`:

The `try` statement allows you to define a block of code to be tested for errors while it is being executed.

The `throw` keyword throws an exception when a problem is detected, which lets us create a custom error.

The `catch` statement allows you to define a block of code to be executed, if an error occurs in the `try` block.

The `try` and `catch` keywords come in pairs:

Example:

```
try
{
    // Block of code to try
    throw exception; // Throw an exception when a problem arise
}
catch ()
{
    // Block of code to handle errors
}
```

Consider the following example:

Example:

```
try
{
    int age = 15;
    if (age >= 18)

        {
            cout << "Access granted - you are old enough.";
        }

    else

        {
            throw (age);
        }
}
catch (int myNum)

{
    cout << "Access denied - You must be at least 18 years old.\n";
    cout << "Age is: " << myNum;
}
```

We use the try block to test some code: If the age variable is less than 18, we will throw an exception, and handle it in our catch block.

In the catch block, we catch the error and do something about it. The catch statement takes a **parameter**: in our example we use an int variable (myNum) (because we are throwing an exception of int type in the try block (age)), to output the value of age.

If no error occurs (e.g. if age is 20 instead of 15, meaning it will be greater than 18), the catch block is skipped:

If you do not know the throw **type** used in the try block, you can use the "three dots" syntax (...) inside the catch block, which will handle any type of exception:

Example:

```
try
{
    int age = 15;
    if (age >= 18)
    {
        cout << "Access granted - you are old enough.";
    }
}
```

```

else
{
    throw 505;
}
}
catch (...)
{
    cout << "Access denied - You must be at least 18 years old.\n";
}

```

```

#include <iostream>
using namespace std;
int main()
{
    double numerator, denominator, divide;

    cout << "Enter numerator : ";
    cin >> numerator;

    cout << "Enter denominator : ";
    cin >> denominator;

    try{
        //throws an exception if denominator is 0
        if(denominator == 0)
        {
            throw 0;
        }

        //Not executed if denominator is 0
        divide = numerator / denominator;
        cout << numerator << " / " << denominator << " = " << divide << endl;
    }
    catch(int num_exception)
    {
        cout << "Error: Cannot divide by " << num_exception << endl;
    }
    return 0;
}

```

**O/P:-**

Enter numerator : 6	Enter numerator : 6
Enter denominator : 0	Enter denominator : 3
Error: Cannot divide by 0	6 / 3 = 2

**Conclusion:** \_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

**Submitted By :**

**Name:**

**Sign:**

**Roll No :**

**Checked By :**

**Name: Ms. Pratiksha S. Patil.**

**Sign:**