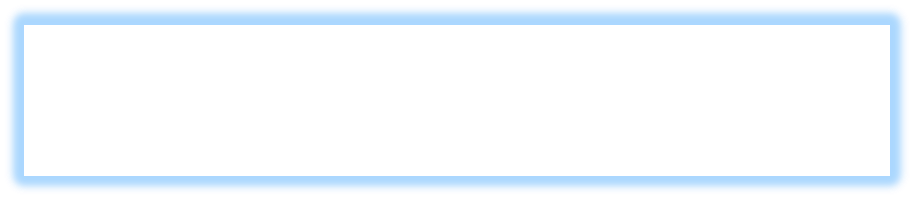
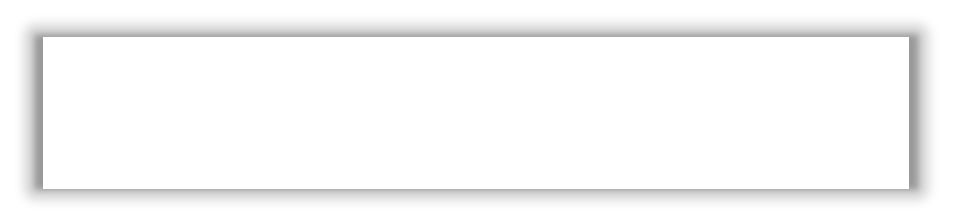
|  |  |
| --- | --- |
| **Unit 4: Polymorphism** | **(9 hrs)** |



**Specific Objectives:**

* Understand and implement the concept of polymorphism in C++.

1. Introduction to Polymorphism
2. Types of Polymorphism: Compile Time Polymorphism: Function Overloading, Operator Overloading Runtime Polymorphism: Virtual Function
3. Overloading Unary and Binary Operators
4. Function Overriding
5. this Pointer and Object Pointer
6. Pure Virtual Function, Abstract Class
7. Virtual Destructor
8. Type Conversion: Basic to User-Defined, User-Defined to Basic, User defined to User- Defined

**1. Introduction to Polymorphism**

Polymorphism is a fundamental concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common superclass. It is based on the idea of "one interface, multiple implementations," where different objects can respond to the same method or function call in different ways.

The term "**polymorphism**" is derived from the Greek word’s "**poly**" (meaning "**many**") and "**morph**" (meaning "**form**"). In the context of programming, *it refers to the ability of objects to take on different forms or behaviors.*

Polymorphism enables code **reusability** and **flexibility** by providing a way to write more generic and generalized code. It allows you to write functions or methods that can operate on objects of different classes, as long as those classes share a common interface or inheritance hierarchy.

There are two main types of polymorphism: **compile-time polymorphism** (also known as

*static or early binding*) and **runtime polymorphism** (also known as *dynamic or late binding*).

### Consider this example:

The “+” operator in C++ can perform two specific functions at two different scenarios i.e. when the “+” operator is used in numbers, it **performs addition**.

int a = 6; int b = 6;

int sum = a + b; *// sum =12*

And the same “+” operator is used in the string, it **performs concatenation**.

string firstName = "Great "; string lastName = "Learning";

*// name = "Great Learning "*

string name = firstName + lastName;

### Key Points:

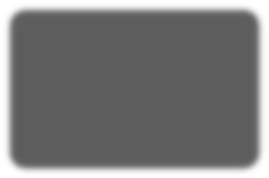
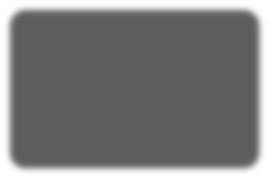
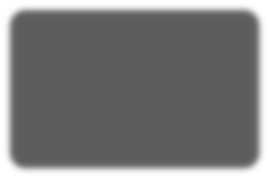
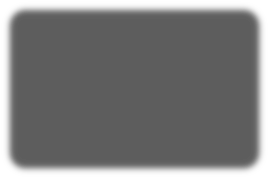
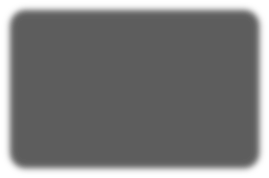
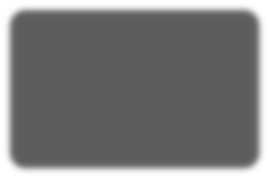
* Polymorphism is one of the main features of object-oriented programming.
* “**Poly**” means **many**, and “**morphism**” details how something has the **ability to change**.
* Polymorphism in C++ means, the same entity (function or object) behaves differently in different scenarios.
* The word “**polymorphism**” means having many forms.
* It is the ability of a message to be displayed in more than one form.
* Polymorphism in C++ allows us to **reuse code** by *creating one function that’s usable for multiple uses*.
* We can also make operators polymorphic and use them to add not only numbers but also combine strings. This saves time and allows for a more streamlined program.
* A real-life example of polymorphism is a person who at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee. So, the same person exhibits different behavior in different situations. This is called polymorphism.



**2. Types of Polymorphism: Compile Time Polymorphism: Function Overloading, Operator Overloading Runtime**

**Polymorphism: Virtual Function**

We can categorize polymorphism into two types. These are Compile-time polymorphism and Run-time polymorphism.



**Polymorphism**

Compile Time /

Static Polymorphism

Run Time /

Dynamic Polymorphism

Function

Overloading

Operator

Overloading

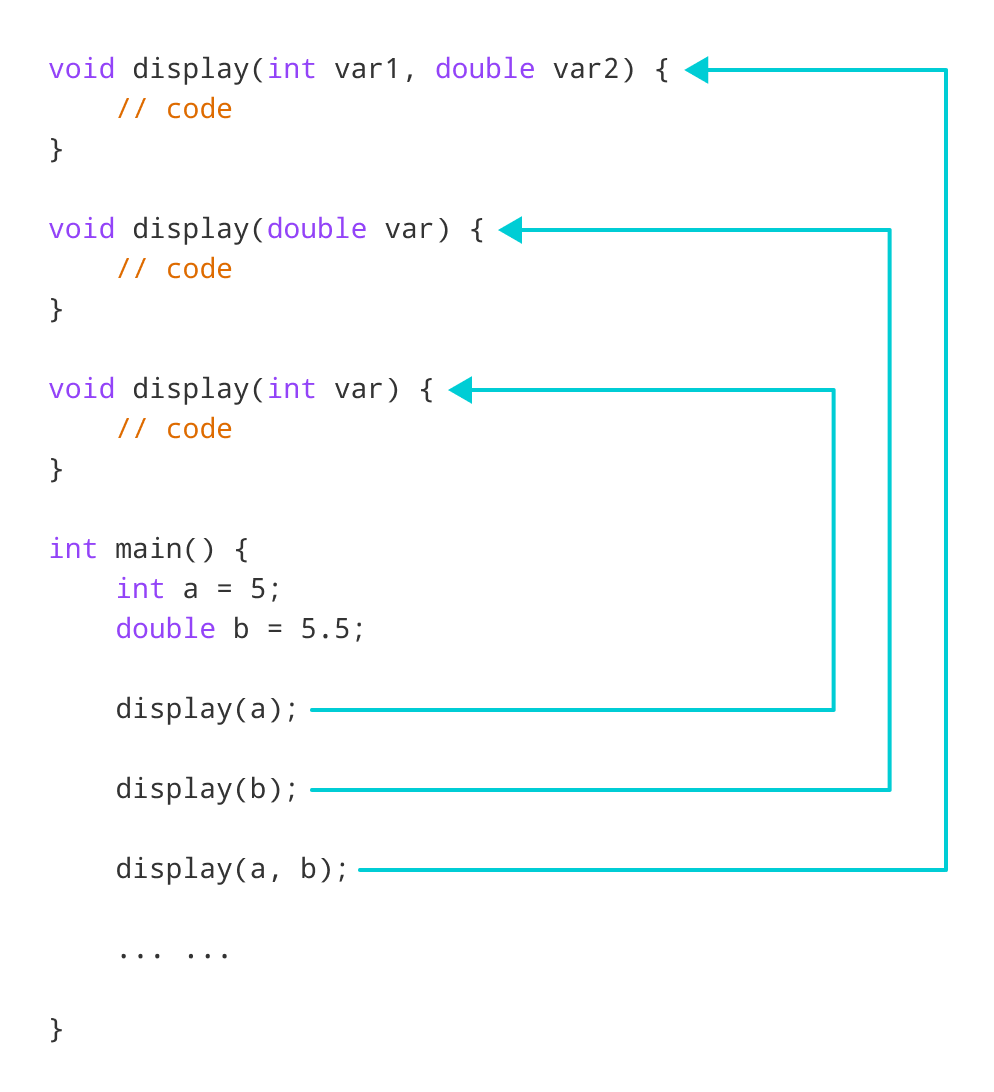
Virtual Function

**Compile Time Polymorphism**

The Compile-time polymorphism is a polymorphism that happens at compile time. What this means is that the compiler must know what is going on. This is also mentioned as static time polymorphism, compile-time binding, static binding, early binding. This type of polymorphism is achieved by function overloading or operator overloading.

### Function Overloading:

Function overloading means one function can perform many tasks. When there are multiple functions with the same name but different parameters, then the functions are said to be **overloaded**, hence this is known as **Function Overloading**. Functions can be overloaded by changing the number of arguments or/and changing the type of arguments. In C++, a single function is used to perform many tasks with the same name and different types of arguments. In the function overloading function will call at the time of program compilation. It is an example of compile-time polymorphism.



### For example:

*// same name different arguments*

int test()

{

*//code*

}

int test(int a)

{

*//code*

}

float test(double a)

{

*//code*

}

int test(int a, double b)

{

*//code*

}

Here, all 4 functions are **overloaded functions**.

Notice that the return types of all these 4 functions are not the same. ***Overloaded functions may or may not have different return types but they must have different arguments***.

### For example,

*// Error code*

int test(int a)

{

*//code*

}

double test(int b)

{

*//code*

}

Here, both functions have the same name, the same type, and the same number of arguments. Hence, the compiler will throw an error.

In the below example, A function ADD() is used to perform two tasks. The two asks would be to add two integer values and add two strings (concatenate). Readability of the program increases by function overloading. It is achieved by using the same name for the same action. #include <iostream>

using namespace std; class Addition

{

public:

int sum;

int ADD(int x,int y) *// Function with parameter*

{

sum=x+y;

return sum; *//this function is performing addition of two Integer value*

}

int ADD() *// Function with same name but without parameter*

{

string a= "NAST";

string b=" College"; *// in this function concatenation is performed*

string c= a+b;

cout<<"The concatenation of a and b is : "<<c<<endl;

}

};

int main()

{

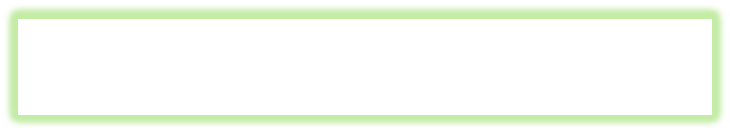
Addition obj; *// Object is created*

cout<<"The addition of x and y is : "<<obj.ADD(10, 20)<<endl; *//first method is called*

obj.ADD(); *// second method is called*

return 0;

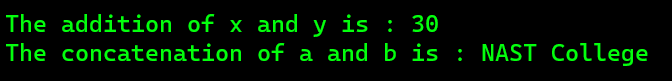
}



**In the above example, we use function ADD() to perform many tasks**

**which is a property of polymorphism in C++.**

### Output:



**Operator Overloading:**

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading in C++ allows you to redefine the behavior of operators for user-defined types. It allows you to use operators such as +, -, \*, /, =, <, etc., with objects of your own classes.

This can make your code more concise and readable, and it can also make it easier to perform operations on user-defined data types that would otherwise be difficult or impossible.

### Syntax for C++ Operator Overloading

To overload an operator, we use a special **operator** function. We define the function inside the class or structure whose objects/variables we want the overloaded operator to work with.

### Syntax 1: within class definition

class className {

... .. ...

public

returnType operator symbol (arguments) {

... .. ...

}

... .. ...

};

Here,

* + *returnType* is the return type of the function.
  + *operator* is a keyword.
  + *symbol* is the operator we want to overload. Like: +, <, -, ++, etc.
  + *arguments* is the arguments passed to the function.

### Syntax 2: outside class definition

returnType className :: operator symbol (arguments)

{

// body of the function.

}

Here,

* + *returnType* is the return type of the function.
  + *className* is the name of the class.
  + *operator* is a keyword.
  + *symbol* is the operator we want to overload. Like: +, <, -, ++, etc.
  + *arguments* is the arguments passed to the function.

### Operator that cannot be overloaded are as follows:

* + Scope operator (::)
  + Sizeof
  + member selector(.)
  + member pointer selector(.\*)
  + ternary operator(?:)

### Rules for Operator Overloading

* + Existing operators can only be overloaded, but the new operators cannot be overloaded.
  + The overloaded operator contains at least one operand of the user-defined data type.
  + We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
  + When **unary operators** are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
  + When **binary operators** are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

For example, you could overload the + operator for a Point class to add two points together. This would allow you to write code like this:

Point p1(1, 2);

Point p2(3, 4); Point p3 = p1 + p2;

Without operator overloading, you would have to write code like this:

Point p3;

p3.x = p1.x + p2.x; p3.y = p1.y + p2.y;

### Example: within class definition

#include <iostream> using namespace std; class A

{

public:

int x;

A(int num)

{

x=num;

}

void operator + (A other)

{

int sum = x+other.x;

cout<<"The addition of two objects is : "<<sum<<endl;

}

};

int main()

{

A a1(5);

A a2(4);

a1+a2; *// a1.operator + (a2);*

return 0;

}

### Example: outside class definition

#include <iostream> using namespace std; class A

{

int x;

public:

A(int num)

{

x=num;

}

void operator+(A);

};

void A :: operator +(A other)

{

int sum = x+other.x;

cout<<"The addition of two objects is : "<<sum<<endl;

}

int main()

{

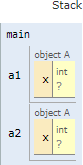
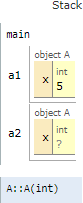
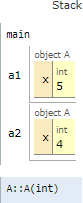
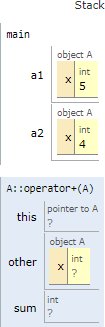
A a1(5);

A a2(4);

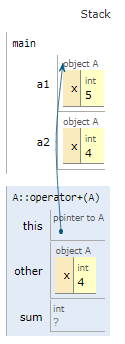
a1+a2; *// a1.operator + (a2);*

return 0;

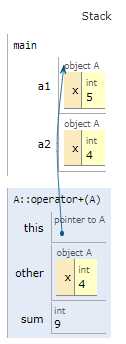
}

main() A a1(5); A a1(4); a1+a2;



**0x61fe1c**



**0x61fe1c**

void operator + (A other) int sum = x+other.x;

**Runtime Polymorphism**

Runtime polymorphism in C++ is a feature that allows the compiler to determine which function to call at runtime, based on the type of object that is being used. This is in contrast to compile-time polymorphism, where the compiler determines which function to call at compile time.

Runtime polymorphism is achieved through the use of virtual functions and function overriding.

**Virtual functions** are declared in the base class with the ***virtual*** keyword. This tells the compiler that the function can be overridden in a derived class. When a virtual function is called, the compiler does not know which function to call until the run time.

**Function overriding** is the process of defining a function in a derived class that has the same name and signature as a function in the base class. When a virtual function is overridden, the derived class function is called instead of the base class function.

### Example:

#include <iostream> using namespace std;

class Shape

{

public:

virtual void draw()

{

cout << "Drawing a Shape." << endl;

}

};

class Circle : public Shape

{

public:

void draw() override

{

cout << "Drawing a Circle." << endl;

}

};

class Rectangle : public Shape

{

public:

void draw() override

{

cout << "Drawing a Rectangle." << endl;

}

};

int main()

{

Shape\* shape1 = new Circle(); Shape\* shape2 = new Rectangle();

shape1->draw(); // Output: Drawing a Circle. shape2->draw(); // Output: Drawing a Rectangle.

return 0;

}

**3. Overloading Unary and Binary Operators**

### Reminder:

* + The mechanism of adding special meaning to an operator is called operator overloading.
  + It provides a flexibility for the creation of new definitions for most C++ operators.
  + Using operator overloading we can give additional meaning to normal C++ operations
  + such as (+,-,=,<=,+= etc.) when they are applied to user defined data types.
  + Usually Operations can perform only on *basic data type (primitive datatype)*. ***Example int a,b,c; c=a+b;*** But if we declare a class complex {} and ***complex c1,c2,c3 ; c3=c1+c2;*** is not possible because *objects are user defined data type(Non-primitive datatype*.
  + Operator overloading helps to define usage of operator for user defined data type i.e. objects.
  + After overloading operands used with operator are objects instead of basic data types.

Operator overloaded function can be invoked using expression such as:

|  |  |  |
| --- | --- | --- |
|  | **In case of member function** | **In case of friend function** |
| **For unary operators** | **op x or x op (e.g.++x)** x.operator op();  ***Note: for postfix x++***  ***Operator op(int)*** | **op x or x op (e.g.++x)** operator op(x)  ***Note: for postfix x++***  ***Operator op(x, int)*** |
| **For binary operators** | **x op y (eg. x+y)**  x.operator op(y); | **x op y (eg x+y)**  operator op(x,y) |

Note:

* + Here op represents the operator being overloaded.
  + x and y represent the object.

# Overloading Unary Operators

A single operand/variable is used with the unary operator to determine the new value of that variable. Unary operators are used with the operand in either the prefix or postfix position. Unary operators come in various forms and have right-to-left associativity and equal precedence.

Unary operators operates on single operand. Some of unary operators are:

1. **Unary plus (+):** This operator is used to indicate a positive value. *For example, if x = 9, then +x will also be 9.*

### Syntax: + operand;

1. **Unary minus (-):** This operator is used to indicate a negative value. *For example, if x = 9, then -x will be -9.*

### Syntax: - operand;

1. **Increment (++):** This operator is used to increase the value of the operand by one. *For example, if x = 9, then ++x will be 10.*

### Syntax:

**++operand; //pre-increment operand++; //post-increment**

1. **Decrement (--):** This operator is used to decrease the value of the operand by one. *For example, if x = 9, then --x will be 8.*

### Syntax:

**--operand; //pre-decrement operand--; //post-decrement**

1. **Logical NOT (!):** This operator is used to get the opposite value of a Boolean expression.

*For example, if x = true, then !x will be false.*

**Syntax: !operand;**

**Unary minus(-) Operator using Member Function:**

The unary minus operator is used to represent negative numbers. For example, if a variable x has the value 7, then -x would have the value -7. A unary operator does not take any argument as it works only on a single operand.

**Example:** #include<iostream> using namespace std;

class Coordinates

{

public:

int X, Y;

*// Constructor to initialize the X and Y*

Coordinates(int x, int y)

{

X = x; Y = y;

}

*// Overloading(-) operator for decrementing X and Y Coordinates*

void operator -()

{

X--;

Y--;

cout <<"Decremented X & Y are " << X << " and " << Y << endl;

}

};

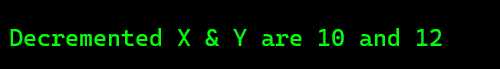
int main()

{

Coordinates C(11,13);

*// Using(-) unary operator for C*

-C; **Output:**

return 0;

}

## Overloading minus(-) Operator using Friend Function

When you overload an operator using a friend function, you can access the private data members of the class, which would not be possible if you overload the operator using a member function, as the subclass cannot access the private members of the superclass.

To overload the minus (-) operator using a friend function, you need to declare the friend function inside the particular class and then define it outside the class. By defining a friend function to overload the minus (-) operator, you can create new and more intuitive syntax for operations involving objects of your class. This can make your code more readable and easier to understand.

**Example:** #include<iostream> using namespace std;

class Coordinates

{

int X, Y; public:

*// Constructor to initialize the X and Y*

Coordinates(int x, int y)

{

X = x; Y = y;

}

friend void operator -(Coordinates &other);

};

*// Overloading(-) operator for decrementing X and Y Coordinates*

void operator -(Coordinates &other)

{

other.X--;

other.Y--;

cout <<"Decremented X & Y are " <<other.X << " and " << other.Y << endl;

}

int main()

{

Coordinates C(11,13);

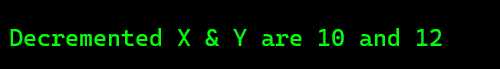
*// Using(-) unary operator for C*

-C;

return 0;

}

**Output:**



## Unary Postfix Increment (++) Operator with return using Member Function:

### Write a simple program to overload (unary ++) postfix operator.

**OR**

### Overloading for postfix increment (++) operator by returning value through object.

#include <iostream> using namespace std; class Counter

{

private:

int count ; public:

void getdata (int x)

{

count=x ;

}

void showdata()

{

cout<<"count="<<count<<endl;

}

Counter operator ++(int)

{

Counter temp; temp.count=count++; return temp;

}

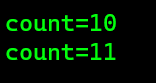
};

int main()

{

Counter c1;

c1.getdata(10); **Output:**

c1.showdata(); c1++;

c1.showdata(); return 0;

}

## Unary Postfix Increment (++) Operator with no return using Member Function:

#include <iostream> using namespace std; class Counter

{

private:

int count ; public:

void getdata (int x)

{

count=x ;

}

void showdata()

{

cout<<"count="<<count<<endl;

}

void operator ++(int)

{

count++;

}

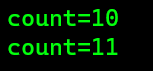
};

int main()

{

Counter c1;

c1.getdata(10); **Output:**

c1.showdata(); c1++;

c1.showdata(); return 0;

}

## Unary Postfix Increment (++) Operator with return using Friend Function:

#include <iostream> using namespace std;

class Counter

{

private:

int count;

public:

void getdata(int x)

{

count = x;

}

void showdata()

{

cout << "count=" << count << endl;

}

friend Counter operator ++(Counter &other, int);

};

Counter operator ++(Counter &other, int)

{

Counter temp = other; other.count++; return temp;

}

int main()

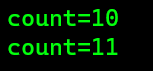
{

Counter c1; c1.getdata(10); c1.showdata(); c1++;

c1.showdata(); return 0;

}

**Output:**



## Unary Postfix Increment (++) Operator with no return using Friend Function:

#include <iostream> using namespace std;

class Counter

{

private:

int count; public:

void getdata (int x)

{

count=x;

}

void showdata()

{

cout<<"count="<<count<<endl;

}

friend void operator ++(Counter &other, int);

};

void operator ++(Counter &other, int)

{

other.count++;

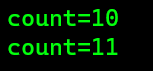
}

int main()

{

Counter c1;

c1.getdata(10); **Output:**

c1.showdata(); c1++;

c1.showdata(); return 0;

}

## Unary Prefix Increment (++) Operator using Member Function:

#include <iostream> using namespace std; class Counter

{

private:

int count; public:

void getdata (int x)

{

count=x;

}

void showdata()

{

cout<<"count="<<count<<endl;

}

Counter operator ++()

{

Counter temp; temp.count=++count; return temp;

}

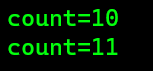
};

int main()

{

Counter c1; c1.getdata(10); c1.showdata();

++c1;

c1.showdata(); **Output:**

return 0;

}

## Unary Prefix Increment (++) Operator using Friend Function:

#include <iostream> using namespace std; class Counter

{

private:

int count; public:

void getdata (int x)

{

count=x;

}

void showdata()

{

cout<<"count="<<count<<endl;

}

friend void operator ++(Counter &other);

};

void operator ++(Counter &other)

{

++other.count;

}

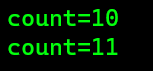
int main()

{

Counter c1; c1.getdata(10); c1.showdata();

++c1;

c1.showdata(); **Output:**

return 0;

}

# Overloading Binary Operators

*(As a theory part we had discussed on the previous topic, please refer that)* **Example 2.1: Add two points value no return values using member function** #include <iostream>

using namespace std; class Point

{

private:

float x; float y;

public:

void input()

{

cout << "Enter points for x and y axis respectively: "<<endl; cin >> x;

cin >> y;

}

*// Overload the + operator*

void operator + (Point obj)

{

int res1 = x + obj.x; int res2 = y + obj.y;

cout << "The New point P3 : " << res1 <<","<< res2 <<endl;

}

};

int main()

{

Point p1, p2;

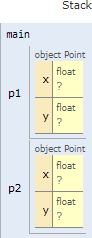
cout << "Enter first Point P1:"<<endl; p1.input();

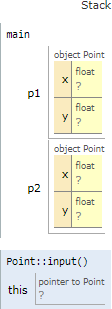
cout << "Enter second Point P2 :"<<endl; p2.input();

p1 + p2; *// p1.operator + (p2);*

return 0;

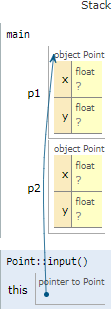
}





**0x61fe10**

**0x61fe18**



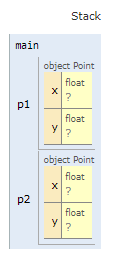
**1**

**2**

**0x61fe18**

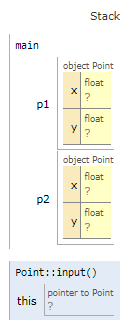
**0x61fe18**

int main() p1.input(); void input()



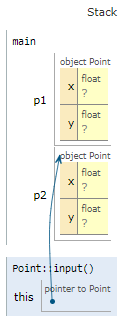
**1**

**2**



**1**

**2**



**0x61fe10**

**0x61fe10**

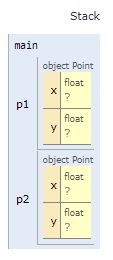
**1**

**2**

**3**

**4**

int main() p2.input(); void input()

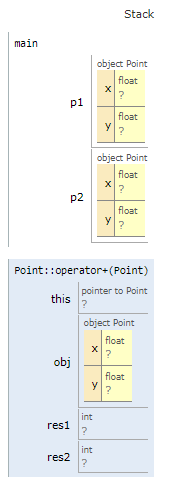


**4**

**3**

**2**

**1**

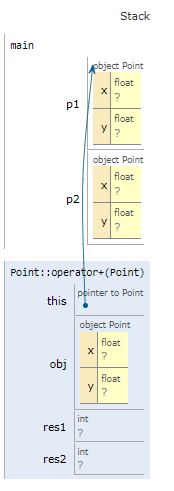


**3**

**1**

**2**

**4**



**4**

**3**

**2**

**1**

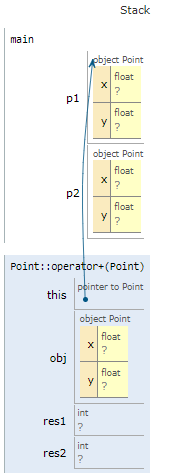
**0x61fe18**

**3**

**0x61fe18**

**4**

int main() p1 + p2; void operator + (Point obj)



**0x61fe18**

**0x61fe18**

**1**

**2**

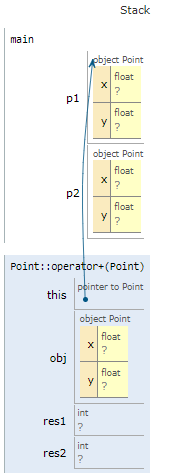
**3**

**4**

**3**

**4**

**4**



**0x61fe18**

**0x61fe18**

**1**

**2**

**3**

**4**

**3**

**4**

**4**

**6**

int res1 = x + obj.x; int res2 = y + obj.y;

### Example2.2: Add two points value with return values using member function

#include <iostream> using namespace std; class Point

{

private:

float x; float y;

public:

void input()

{

cout << "Enter points for x and y axis respectively: "<<endl; cin >> x;

cin >> y;

}

*// Overload the + operator*

Point operator + (Point obj)

{

Point temp;

temp.x = x + obj.x; temp.y = y + obj.y; return temp;

}

void output()

{

cout << "The New point P3 : " << x <<","<< y <<endl;

}

};

int main()

{

Point p1, p2, p3;

cout << "Enter first Point P1:"<<endl; p1.input();

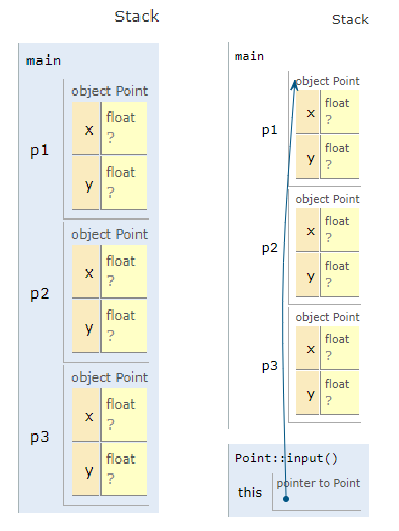
cout << "Enter second Point P2 :"<<endl; p2.input();

p3 = p1 + p2; *// p3 = p1.operator + (p2);*

p3.output();

return 0;

}

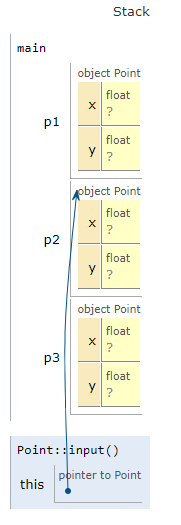


**0x61fe18**

**1**

**2**

**0x61fe18**



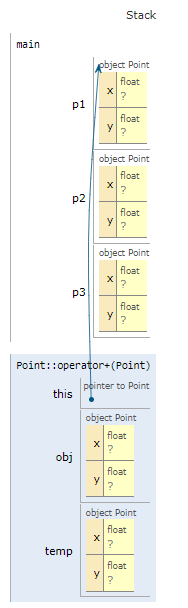
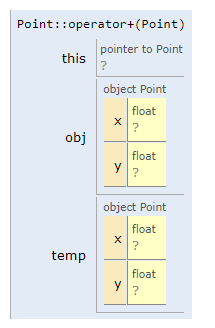
**0x61fe10**

**3**

**4**

**0x61fe10**

Point p1, p2, p3; p1.input(); p2.input();



**0x61fe18**

**0x61fe08**

**3**

**4**

**3**

**4**

**4**

**6**

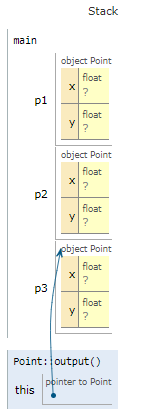
**4**

**3**

**2**

**1**

**0x61fe18**



**4**

**3**

**2**

**1**

**0x61fe08**

**0x61fe08**

**4**

**6**

p3 = p1 + p2; Point operator + (Point obj) p3.output();

* 1. What do you mean by operator overloading? How do your overload the + operator in main program (c3=c1+c2).so that c3 can store a complex number obtained by adding c1 and c2.
  2. What is polymorphism? How operator overloading is used to support polymorphism. Explain it by overloading ‘+’ operator to concatenate two strings.

### Example 2.3: Add two points value with no return values using friend function

#include <iostream> using namespace std; class Point

{

private:

float x; float y;

public:

void input()

{

cout << "Enter points for x and y axis respectively: " << endl; cin >> x;

cin >> y;

}

friend void operator+(Point obj1, Point obj2);

};

void operator+(Point obj1, Point obj2)

{

float res1 = obj1.x + obj2.x; float res2 = obj1.y + obj2.y;

cout << "The New point P3 : " << res1 << ", " << res2 << endl;

}

int main()

{

Point p1, p2;

cout << "Enter first Point P1:" << endl; p1.input();

cout << "Enter second Point P2 :" << endl; p2.input();

p1 + p2; *// operator+(p1, p2);*

return 0;

}

### Example 2.4: Add two points value with return values using friend function

#include <iostream> using namespace std;

class Point

{

private:

float x; float y;

public:

void input()

{

cout << "Enter points for x and y axis respectively: " << endl; cin >> x;

cin >> y;

}

friend Point operator+(Point obj1, Point obj2);

void display()

{

cout << "The New point P3: " << x << ", " << y << endl;

}

};

Point operator+(Point obj1, Point obj2)

{

Point result;

result.x = obj1.x + obj2.x; result.y = obj1.y + obj2.y; return result;

}

int main()

{

Point p1, p2, p3;

cout << "Enter first Point P1:" << endl; p1.input();

cout << "Enter second Point P2 :" << endl; p2.input();

p3 = p1 + p2; *// operator+(p1, p2);*

p3.display(); return 0;

}

Exam Reference only || Incomplete Note