CS-1004 Object Oriented programming
Week 10

Instructor:
ATIYA
LECTURER
(COMPUTER SCIENCE DEPARTMENT)
NATIONAL UNIVERSITY- FAST (KHI CAMPUS)
EMAIL: ATIYA.JOKHIO@NU.EDU.PK
ROOM: (LECTURER ROOM#19, CS DEPT BASEMENT-2)

Operator Overloading

- C++ allows you to specify more than one definition for an operator in the same scope, which is called operator overloading.
- You can redefine or overload most of the built-in operators available in C++
- It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it.

Operator Overloading

- Defining a new behavior for common operators of a language
- C++ enables you to overload most operators to be sensitive to the context in which they’re used
- Using operator overloading makes a program clearer than accomplishing the same operations with function calls

When operators are overloaded as member functions, they must be non-static

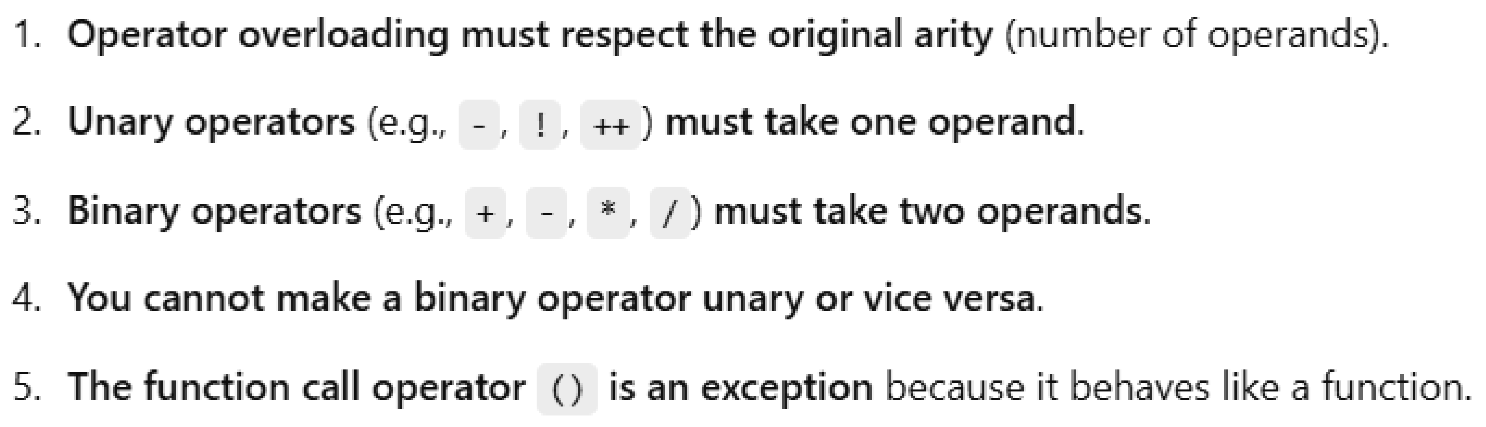
- When operators are overloaded as **member functions**, they must be **non-static** because they operate on **instances** of a class
- Static Functions Do Not Have this
- A static function does not have access to this, meaning it cannot directly access instance variables.
- Since operator overloading usually involves modifying or accessing object attributes, a static function would not work properly.

Operator Overloading

- An operator is overloaded by writing a non-static member function definition or global function definition
- When operators are overloaded as member functions, they must be non-static
- To use an operator on class objects (as operands), that operator **“must” be overloaded**

Operator Overloading

- Operator overloading cannot change the arity of an operator
- Operator overloading works when *at least* one argument of that operator is an object
- We cannot create new operators using operator overloading

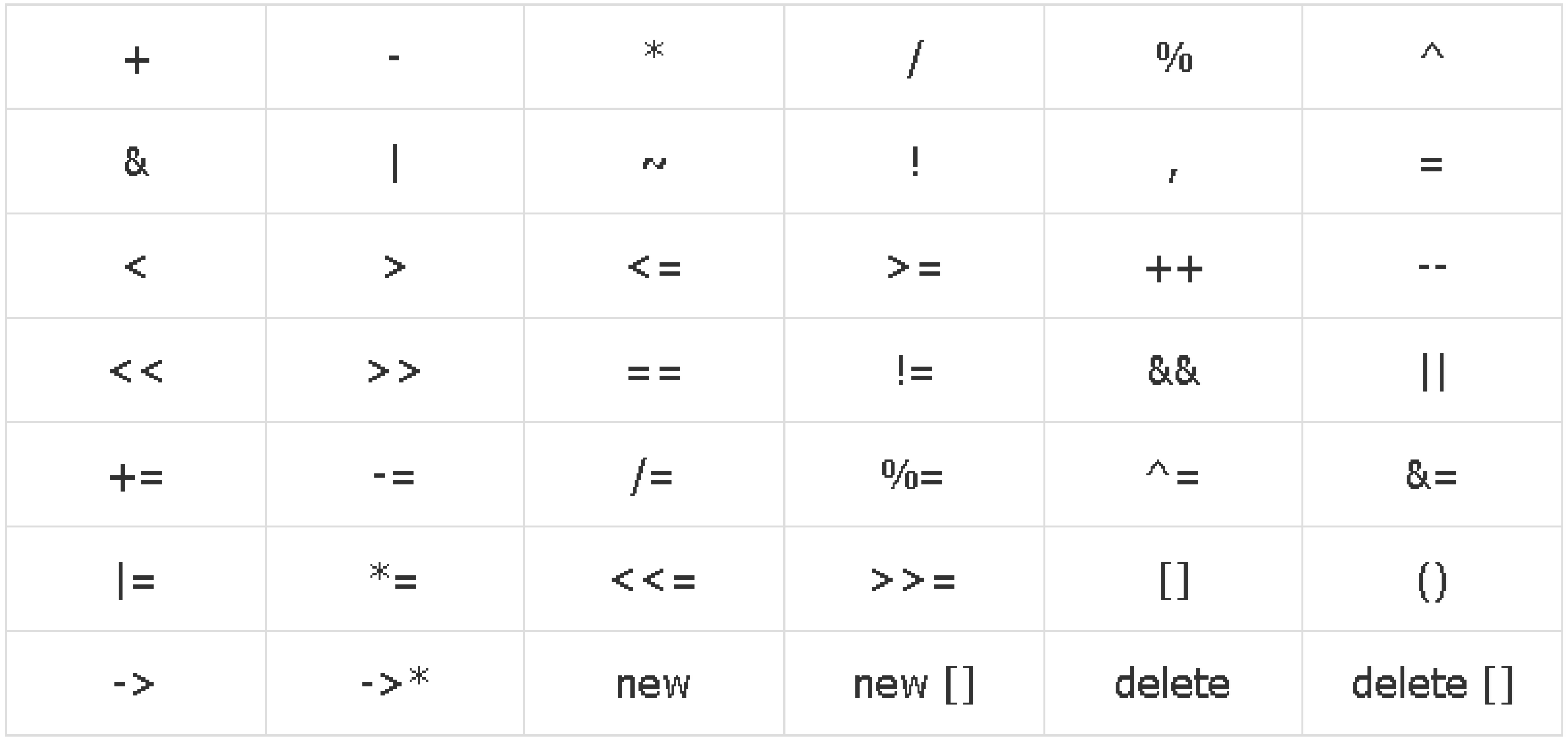


1. **Operator overloading must respect the original arity** (number of operands).
2. **Unary operators** (e.g., `-`, `!`, `++`) **must take one operand.**
3. **Binary operators** (e.g., `+`, `-`, `*`, `/`) **must take two operands.**
4. **You cannot make a binary operator unary or vice versa.**
5. **The function call operator `()` is an exception** because it behaves like a function.

LIST OF OPERATORS THAT CAN’T BE OVERLOADED

- Almost any operator can be overloaded in C++. However there are few operator which can not be overloaded.
- scope operator (::)
- sizeof
- member selector –(.)
- member pointer selector – (.*)
- ternary operator – (?:)

Overloadable Operators



+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

BUILT IN OVERLOADS

Most operators are already overloaded for fundamental types.

Example:

1) In the case of the expression:

a / b the operand type determines the machine code created by the compiler for the division operator. If both operands are integral types, an integral division is performed;

in all other cases floating-point division occurs. Thus, different actions are performed depending on the operand types involved.

2) <<, which is used both as the stream insertion operator and as the bitwise left-shift operator.

Works fine

```
#include <iostream>
using namespace std;
int main() {
        int a=5;
        int b=3;
        int z=a+b;
        cout << z;
   return 0;}
```

Output?

```
#include <iostream>
using namespace std;
 class Complex {
   private:
      int real;
      int image;
   public:
Complex(){
         real = 0;
         image = 0; }
Complex(int r, int i){
         real = r;
         image = i;}
```

Output?

```
 void displayComplex() {
 cout << "real: "<< real << " Imaginary:" <<image <<endl;     }};
int main() {
            Complex c1(2,1);
            Complex c2(3,1);
            Complex c3;
            c3=c1+c2;
   return 0;}
```

Output

```
[Error] no match for 'operator+' (operand types are 'Complex' and 'Complex')
```

criteria/rules to define the operator function:

- In case of a non-static function, the binary operator should have only one argument and unary should not have an argument.

```
// Overloading unary '-'
    Number operator-() {
        return Number(-value);
    }
 // Overloading binary '+'
    Number operator+(Number other) {
        return Number(value + other.value);
    }
```

Syntax of Operator Overloading

Return type operator op(argument list);

```
return_type class_name  : : operator op(argument_list)
{
// function body
}
```

Example of Operator Overloading

```
class Complex {
  private:
    int real;
    int image;
  public:
Complex(){
      real = 0;
      image = 0; }
Complex(int r, int i){
      real = r;
      image = i;}
```

Example of Operator Overloading

```
void displayComplex() {
 cout << "real: "<< real << " Imaginary:" <<image <<endl;     }
// overloaded  (+) operator
     Complex operator+ (Complex c) {
             Complex temp;
        temp.real=real+c.real;
        temp.image=image+c.image;
        return temp;
     }};
```

Example of Operator Overloading

```
int main() {
            Complex c1(2,1);
            Complex c2(3,1);
            Complex c3;
         c3 = c1.operator+ ( c2 );
          //c3=c1+c2;
         c3.displayComplex();
    return 0;}
```

c3=c1+c2; // statement 1

It is same like

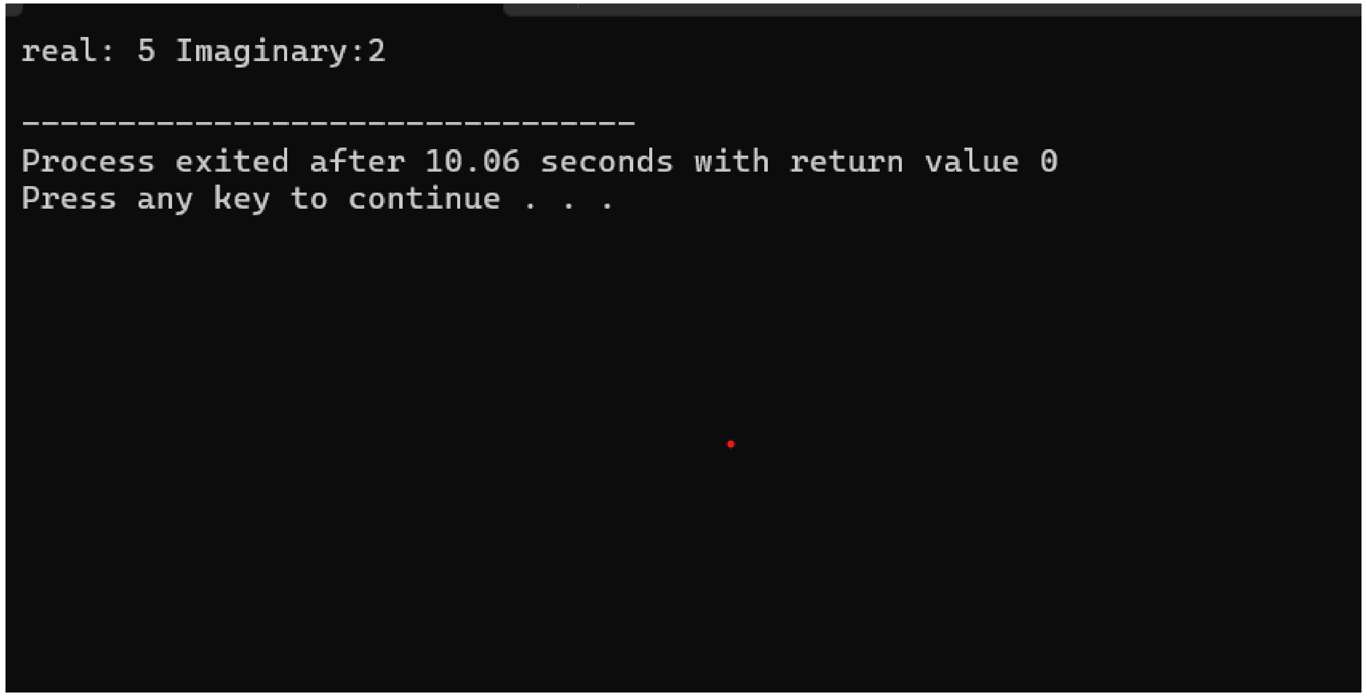
c3=c1.add(c2); just for understanding

In statement 1, Left object c1 will invoke operator+() function and right object c2 is passing as an argument.

Another way of calling binary operator overloading function is to call like a normal member function as follows,

c3 = c1.operator+ (c2);

Output



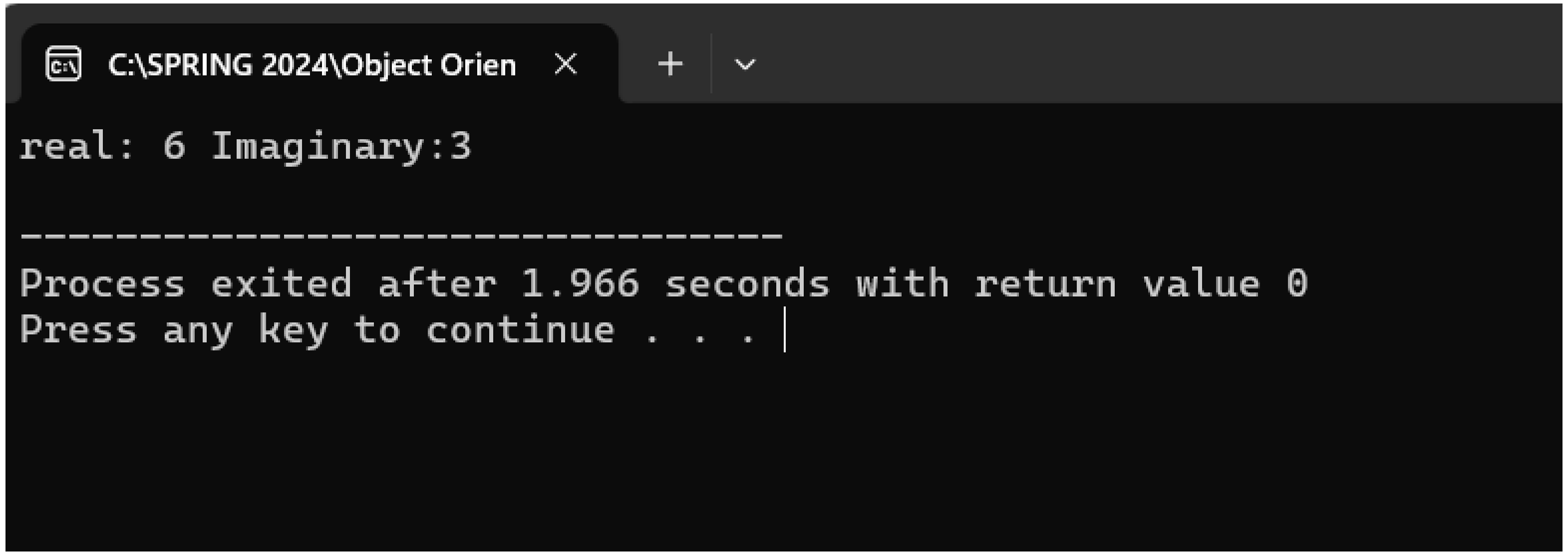
```
real: 5 Imaginary:2

--------------------------------
Process exited after 10.06 seconds with return value 0
Press any key to continue . . .
```

```
int main() {
            Complex c1(2,1);
            Complex c2(3,1);
            Complex c3(1,1);
            Complex c4;
            c4=c1+c2+c3;
            c4.displayComplex();
   return 0;}
```

c4=c1+c2+c3;

Firstly addition of c2+c3 happen then the result of c2+c3 added to c1 and stored in c4 here add function called two times

Output

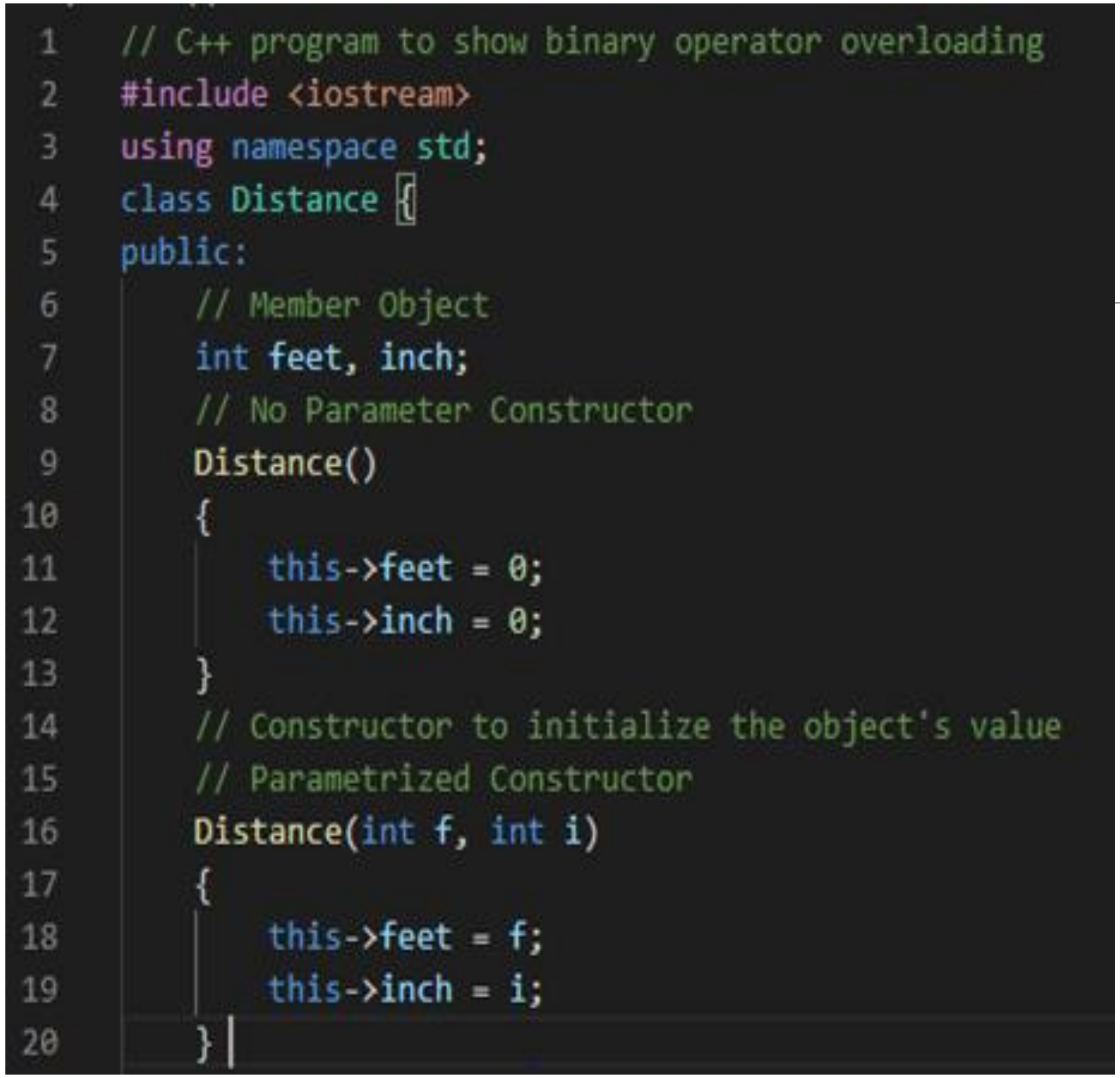


```
real: 6 Imaginary:3

--------------------------------
Process exited after 1.966 seconds with return value 0
Press any key to continue . . .
```

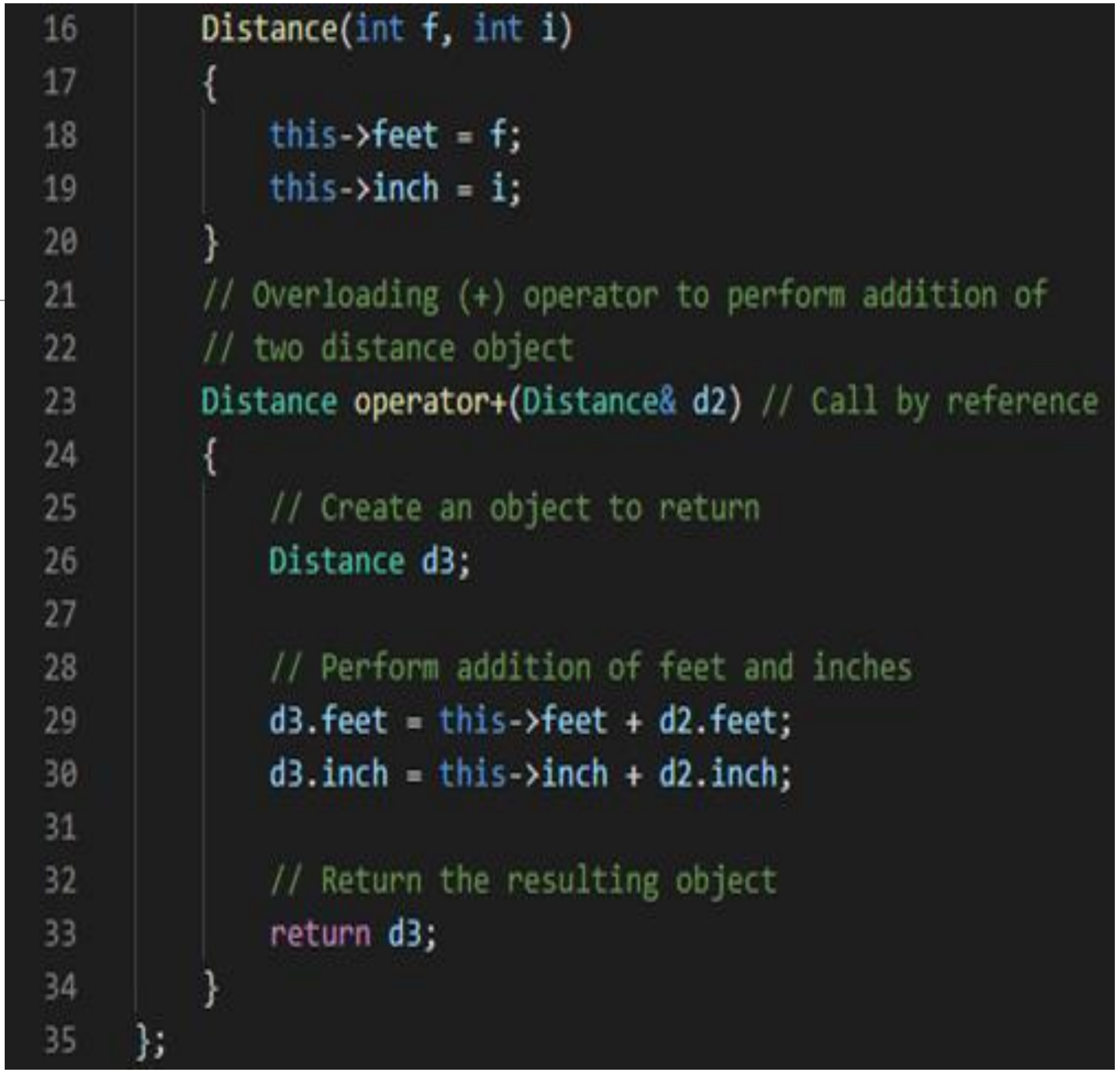
Binary Operator

- In binary operator overloading function, there should be one argument to be passed. It is overloading of an operator operating on two operands.
- Let’s take the same example of class Distance, but this time, add two distance objects.

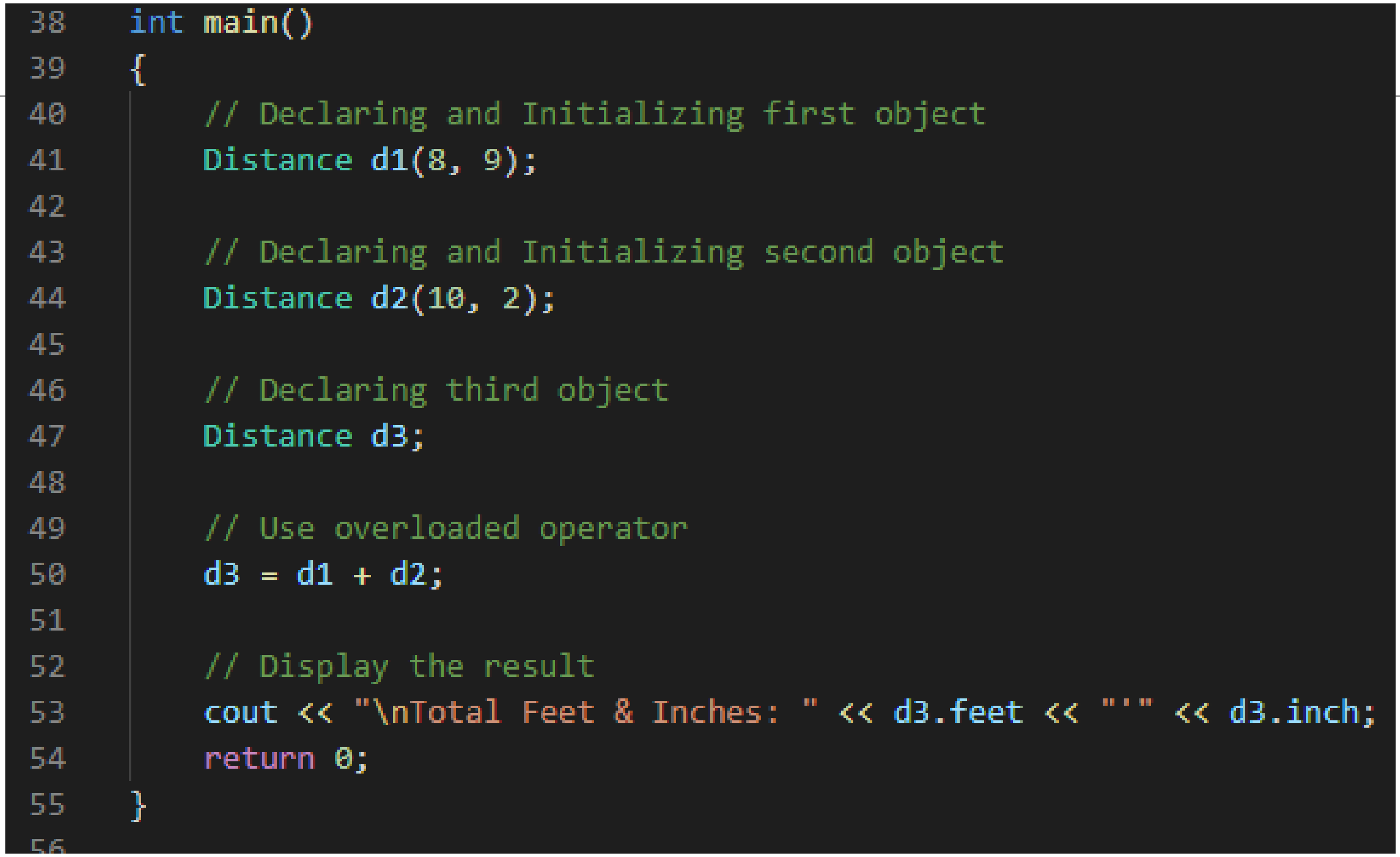
```
d3 = d1 + d2;
```



```cpp
// C++ program to show binary operator overloading
#include <iostream>
using namespace std;
class Distance {
public:
    // Member Object
    int feet, inch;
    // No Parameter Constructor
    Distance()
    {
        this->feet = 0;
        this->inch = 0;
    }
    // Constructor to initialize the object's value
    // Parametrized Constructor
    Distance(int f, int i)
    {
        this->feet = f;
        this->inch = i;
    }
```



```cpp
    Distance(int f, int i)
    {
        this->feet = f;
        this->inch = i;
    }
    // Overloading (+) operator to perform addition of
    // two distance object
    Distance operator+(Distance& d2) // Call by reference
    {
        // Create an object to return
        Distance d3;

        // Perform addition of feet and inches
        d3.feet = this->feet + d2.feet;
        d3.inch = this->inch + d2.inch;

        // Return the resulting object
        return d3;
    }
};
```



```cpp
int main()
{
    // Declaring and Initializing first object
    Distance d1(8, 9);

    // Declaring and Initializing second object
    Distance d2(10, 2);

    // Declaring third object
    Distance d3;

    // Use overloaded operator
    d3 = d1 + d2;

    // Display the result
    cout << "\nTotal Feet & Inches: " << d3.feet << "'" << d3.inch;
    return 0;
}
```

```
#include <iostream>
using namespace std;
class Binary {
    int i;
public:
    Binary(int val = 0) : i(val) {}

    // Binary + Operator Overload (returns new object)
    Binary operator+(const Binary& obj) {
        Binary temp;  // Create a new object (temp)
        temp.i = this->i + obj.i;
        return temp;  // Return new object (not reference)
    }
    void Display() const {
        cout << "i=" << i << endl;
    }
};
```

```
int main() {
   Binary obj1(5), obj2(3), obj3;

   obj3 = obj1 + obj2;  // obj3 = obj1.operator+(obj2);
   obj3.Display();  // Output: i=8

   Binary obj4;
   obj4 = obj1 + obj2 + obj3;  // Chaining works: obj4 = (obj1 + obj2) + obj3;
//obj4 = obj1.operator+(obj2).operator+(obj3);

   obj4.Display();  // Output: i=16

   return 0;
}
```

Step-by-Step Execution of obj1 + obj2 + obj3

1. First, obj1 + obj2
   - temp1.i = obj1.i + obj2.i = 5 + 3 = 8
   - Returns a new object (temp1 with i=8).
2. Then, temp1 + obj3
   - temp2.i = temp1.i + obj3.i = 8 + 8 = 16
   - Returns another new object (temp2 with i=16).
3. Assign result to obj4
   - obj4 = temp2
   - obj4.i = 16

Step-by-Step Breakdown:

1. obj1 + obj2 → Calls obj1.operator+(obj2), returns a new object (say temp1).
2. temp1 + obj3 → Calls temp1.operator+(obj3), returns another new object (say temp2).
3. obj4 = temp2 → Assigns temp2 to obj4.

Unary Operator

- The unary operators operate on a single operand and following are the examples of Unary operators −
  - The increment (++) and decrement (--) operators.
  - The unary minus (-) operator.
  - The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--.

Prefix Increment (++i)

- Increments first, then returns the updated value.

```
#include <iostream>
int main() {
    int i = 5;
    int x = ++i;

    std::cout << "i = " << i << std::endl;
    std::cout << "x = " << x << std::endl;}
```

Postfix Increment (i++)

- Returns the original value first, then increments.

```
#include <iostream>
int main() {
    int i = 5;
    int x = i++;

    std::cout << "i = " << i << std::endl;
    std::cout << "x = " << x << std::endl;
}
```

Difference between post and prefix

```
int y=0;

y=x++;

cout <<x;

cout<<y;
```

X=1

Y=0

In postfix firstly substitute the value then increment happens

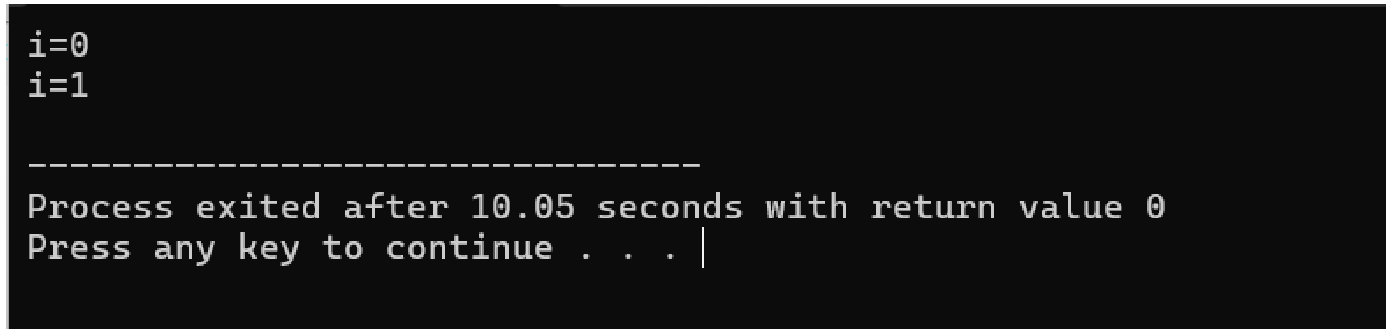
For Prefix ++ operator

```
void operator ++ ( )
{
        ++x;
        ++y;
}
```

(Works the same way for prefix decrement operator)

For Prefix ++ operator

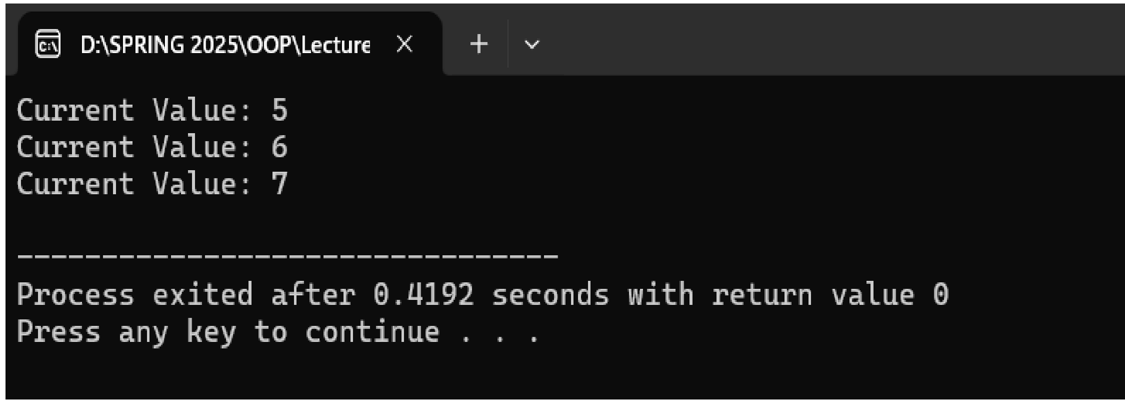
```
class Prefix{
      int i;
   public:
      Prefix(): i(0) {  }
      void operator ++()
        { ++i; }
      void Display()
        { cout << "i=" << i << endl; }};
int main(){
   Prefix obj;
   obj.Display();
   ++obj;
   //you can also write  obj.operator ++();
   obj.Display();
              return 0;}
```



```
i=0
i=1

--------------------------------
Process exited after 10.05 seconds with return value 0
Press any key to continue . . .
```

For Prefix ++ operator

```
#include <iostream>
class Number {
    int value;
public:
    Number(int v) : value(v) {}
    Number(){    }
    Number operator++() {
        Number z;
    z= ++value;
        return z;  }
    void display() const {
std::cout << "Current Value: " << value << std::endl;
    }};
int main() {
    Number obj(5);
    obj.display();
    Number o;
    o=++(++obj);  //obj.operator++().operator++();
    obj.display();
    o.display();//effects on z object
    return 0;
}
```



```
Current Value: 5
Current Value: 6
Current Value: 7

--------------------------------
Process exited after 0.4192 seconds with return value 0
Press any key to continue . . .
```

distinguish between post and prefix

- The operator symbol for both **prefix(++i)** and **postfix(i++)** are the same.
- Hence, we need two different function definitions to distinguish between them.
- This is achieved by passing a dummy int parameter in the postfix version.

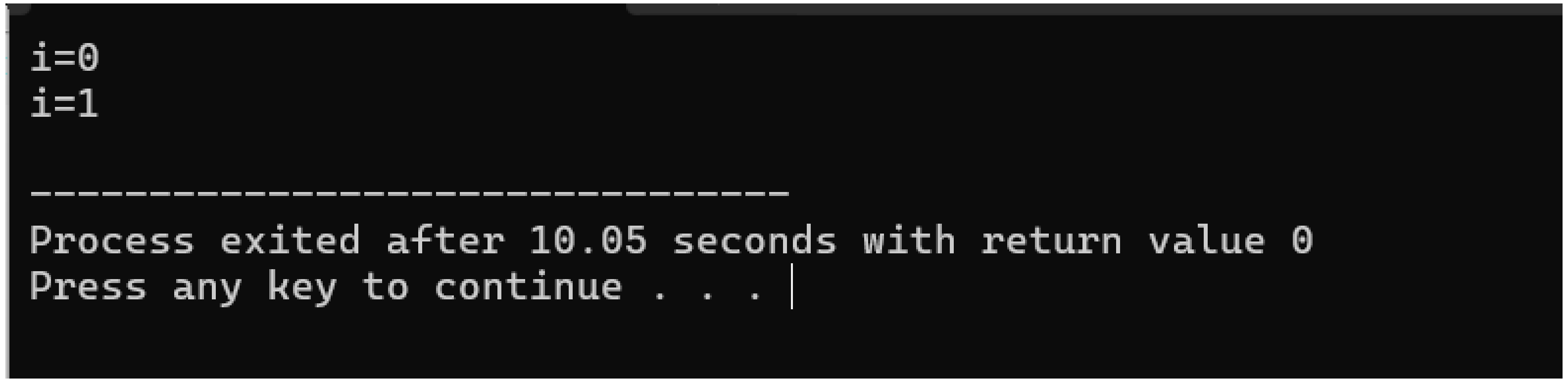
For Postfix ++ Operator

```
Vector operator ++ ( int )
{
        Vector temp;
        temp.x = x++;
        temp.y = y++;
        return temp;
}
```

For Postfix ++ Operator

```
#include<iostream>
using namespace std;
class Postfix{
      int i;
   public:
      Postfix(): i(0) {  }
      void operator ++(int)
         { i++; }
      void Display()
         { cout << "i=" << i << endl; }};
int main(){
   Postfix obj;
   obj.Display();
                  obj.operator ++(4);//for this syntax you must have to pass a dummy value
                  // you can also write obj++;
   obj.Display();
                   return 0;}
```

Output



```
i=0
i=1

--------------------------------
Process exited after 10.05 seconds with return value 0
Press any key to continue . . .
```

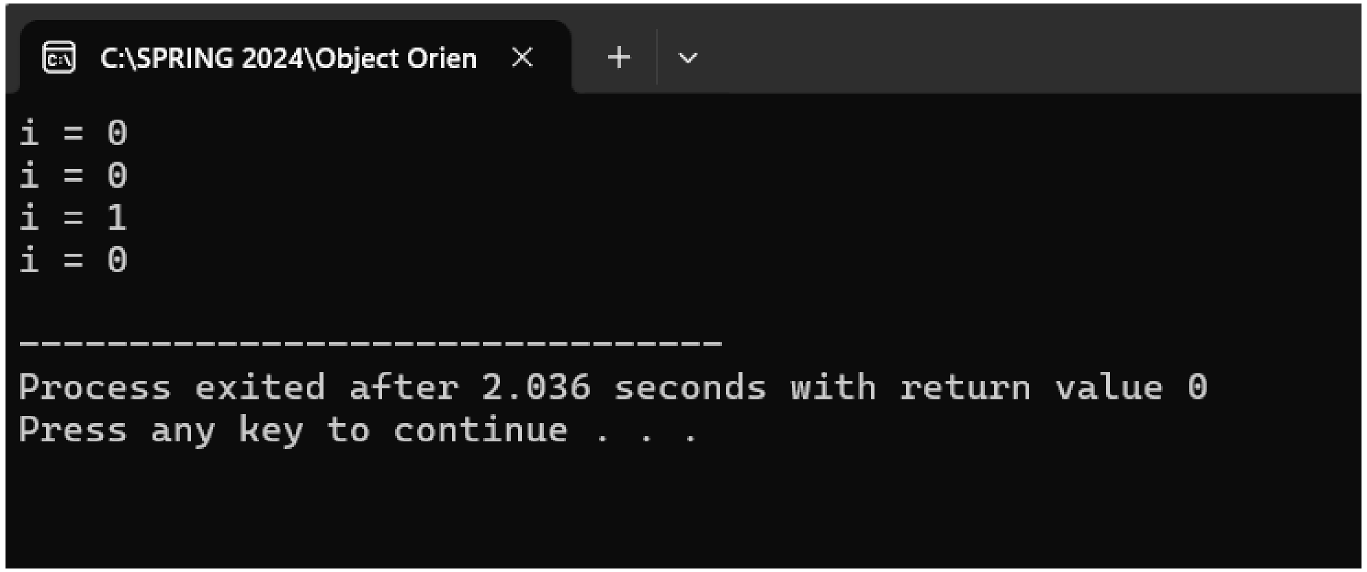
Output?

```
class Check{
    int i;
  public:
    Check(): i(0) {  }
    Check operator ++ (int){
        Check temp;
        temp.i = i++;
        return temp;}
void Display()
    { cout << "i = "<< i <<endl; }};
```

Output?

```
int main(){
   Check obj, obj1;
   obj.Display();
   obj1.Display();
   obj1 = obj++;
   obj.Display();
   obj1.Display();
   return 0;}
```

Output



```
i = 0
i = 0
i = 1
i = 0

--------------------------------
Process exited after 2.036 seconds with return value 0
Press any key to continue . . .
```

Assignment Operator =

The assignment operator has a signature like this:

```
class MyClass {
public:
MyClass & operator=(const MyClass &rhs);
} ;
int main(){
MyClass a, b; ...
b = a; // Same as b.operator=(a);
}
```

- Notice that the = operator takes a const-reference to the right hand side of the assignment.
- The reason for this should be obvious, since we don't want to change that value; we only want to change what's on the left hand side.
- Also, you will notice that a reference is returned by the assignment operator.
-
- This is to allow **operator chaining**. You typically see it with primitive types, like this:

operator chaining

operator chaining with primitive types, like this:

int a, b, c, d, e;

a = b = c = d = e = 42;

This is interpreted by the compiler as:

a = (b = (c = (d = (e = 42))));

In other words, assignment is right-associative. The last assignment operation is evaluated first, and is propagated leftward through the series of assignments. Specifically:

e = 42 assigns 42 to e, then returns e as the result

The value of e is then assigned to d, and then d is returned as the result

The value of d is then assigned to c, and then c is returned as the result etc.

operator chaining

Now, in order to support operator chaining, the assignment operator must return some value. The value that should be returned is a reference to the left-hand side of the assignment.

Operator Functions as Class Members vs. Global Functions

- Operator functions can be member functions(already discussed)

or global functions (non-member function)

- global functions are often made friends for performance reasons.

performance reasons?

performance reasons

It's possible to overload an operator as a global, non-friend function, but such a function requiring access to a class's private or protected data would need to use set *or* get *functions provided in that class's public interface. The overhead of calling these functions could cause poor performance, so these functions can be inlined to improve performance.*

Operator Overloading using a Friend function

In this approach, the operator overloading function must precede with friend keyword, and declare a function class scope. Keeping in mind, friend operator function takes two parameters in a binary operator, varies one parameter in a unary operator. All the working and implementation would same as binary operator function except this function will be implemented outside of the class scope.

Operator Overloading using a Friend function

- In case of a non-static function, the binary operator should have only one argument and unary should not have an argument.
- In the case of a friend function, the binary operator should have only two argument and unary should have only one argument.

Syntax for binary operator overloading using friend function

```
friend return-type operator operator-symbol (Variable
1, Varibale2)
{
    //Statements;
}
```

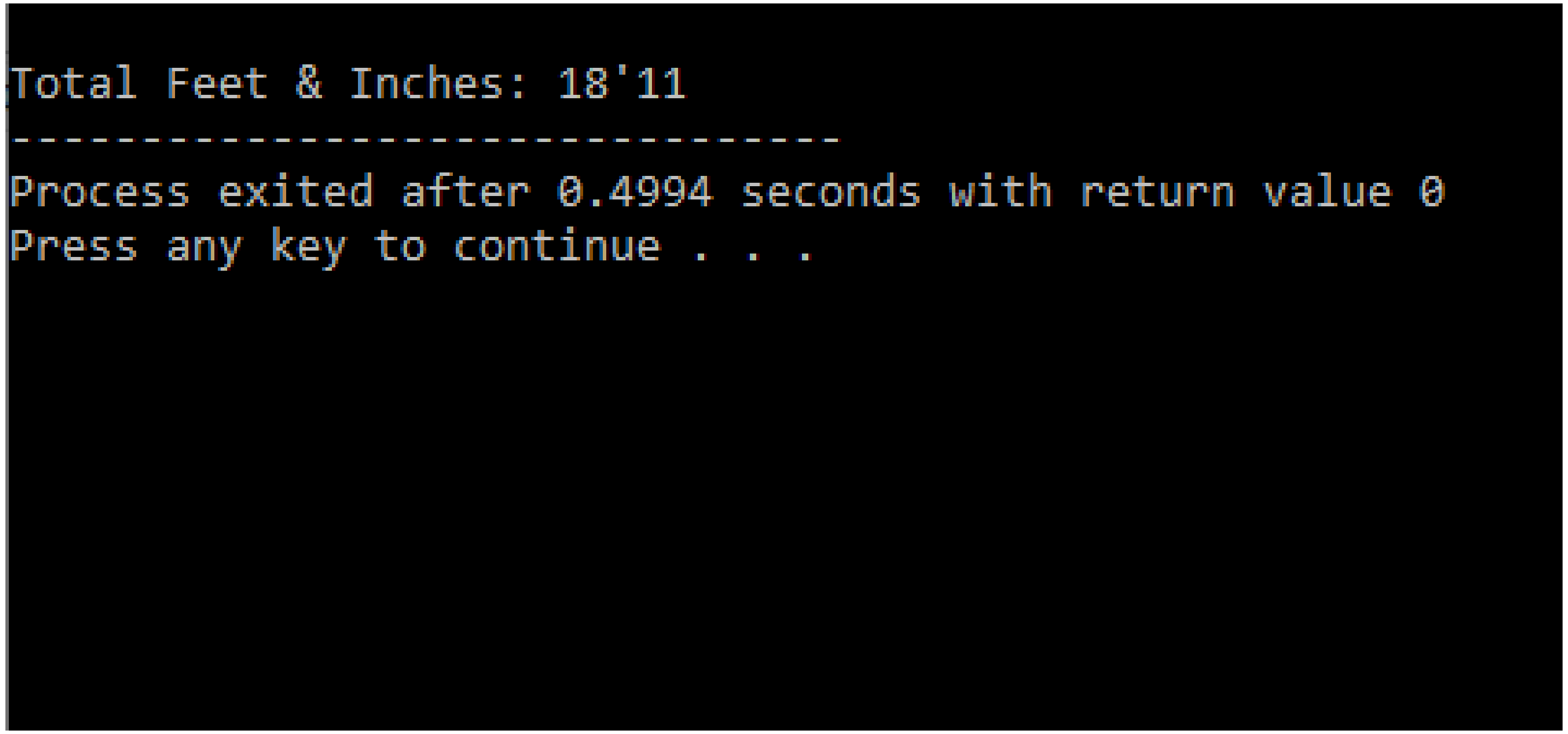
Overloading Binary Operator using a Friend function

```
class Distance {
public:
  int feet, inch;
    Distance()  {
        this->feet = 0;
        this->inch = 0;  }
    Distance(int f, int i)   {
        this->feet = f;
        this->inch = i;  }
    friend Distance operator+(Distance, Distance);  };
```

Overloading Binary Operator using a Friend function

```
Distance operator+(Distance d1, Distance d2)  {
    Distance d3;
    d3.feet = d1.feet + d2.feet;
    d3.inch = d1.inch + d2.inch;
    return d3; }
int main() {
    Distance d1(8, 9);
    Distance d2(10, 2);
    Distance d3;
    d3 = d1 + d2;
    cout << "\nTotal Feet & Inches: " << d3.feet << "'" << d3.inch;
    return 0; }
```

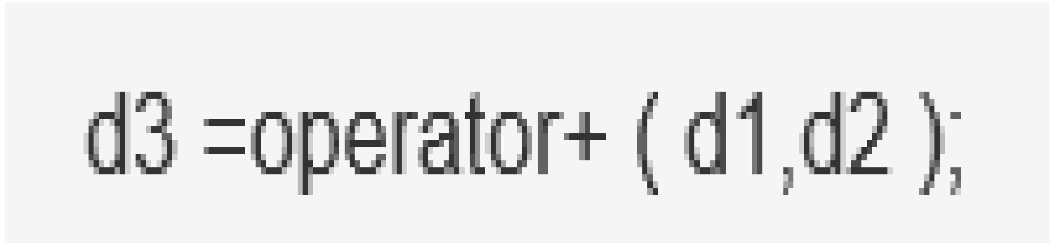
Overloading Binary Operator using a Friend function



```
Total Feet & Inches: 18'11
--------------------------------
Process exited after 0.4994 seconds with return value 0
Press any key to continue . . .
```

Overloading Binary Operator using a Friend function

Another way of calling binary operator overloading with friend function is to call like a non-member function as follows,



```
d3 =operator+ ( d1,d2 );
```

Unary operator overloading using Friend function

```
class UnaryFriend{
    int a=10;
    int b=20;
    public:
        void getvalues(){
            cout<<"Values of A and B\n";
            cout<<a<<"\n"<<b<<"\n"<<endl;}
        void friend operator-(UnaryFriend &x); };
void operator-(UnaryFriend &x){
    x.a = -x.a;    //Object name must be used as it is a friend function
    x.b = -x.b;}
```

Unary operator overloading using Friend function

```
int main(){
    UnaryFriend x1;
    cout<<"Before Overloading\n";
    x1.getvalues();
    cout<<"After Overloading \n";
    -x1;// operator-(x1);
     x1.getvalues();
     return 0;}
```

To distinguish between prefix and postfix add a dummy int datatype in postfix

E.g.

Friend void operator--(UnaryFriend x , int dummy)

some criteria/rules to define the operator function:

Operator overloading function can be applied on a member function if the left operand is an object of that class, but if the Left operand is different, then the Operator overloading function must be defined as a non-member function.

When to use a member function for operator overloading?

•If the left operand of the operator is an object of the same class,
the operator overloading function can be implemented as a member function.
•It automatically passes the left operand (this) to the function.

When to use a non-member (friend) function?

•If the left operand is not an object of the class, we must define the operator overloading function as
a non-member function (usually a friend function) so that it can access private members.

Member Function (Left Operand is an Object of the Class)

```
#include <iostream>
using namespace std;
class Number {
    int value;
public:
    Number(int v) : value(v) {}
// Overloading + operator as a member function
    Number operator+(const Number& obj) {
        return Number(value + obj.value);  }
void display() { cout << "Value: " << value << endl; }};
int main() {
    Number n1(10), n2(20);
    Number result = n1 + n2; // Works because n1 is an object of Number
    result.display();
    return 0;}
```

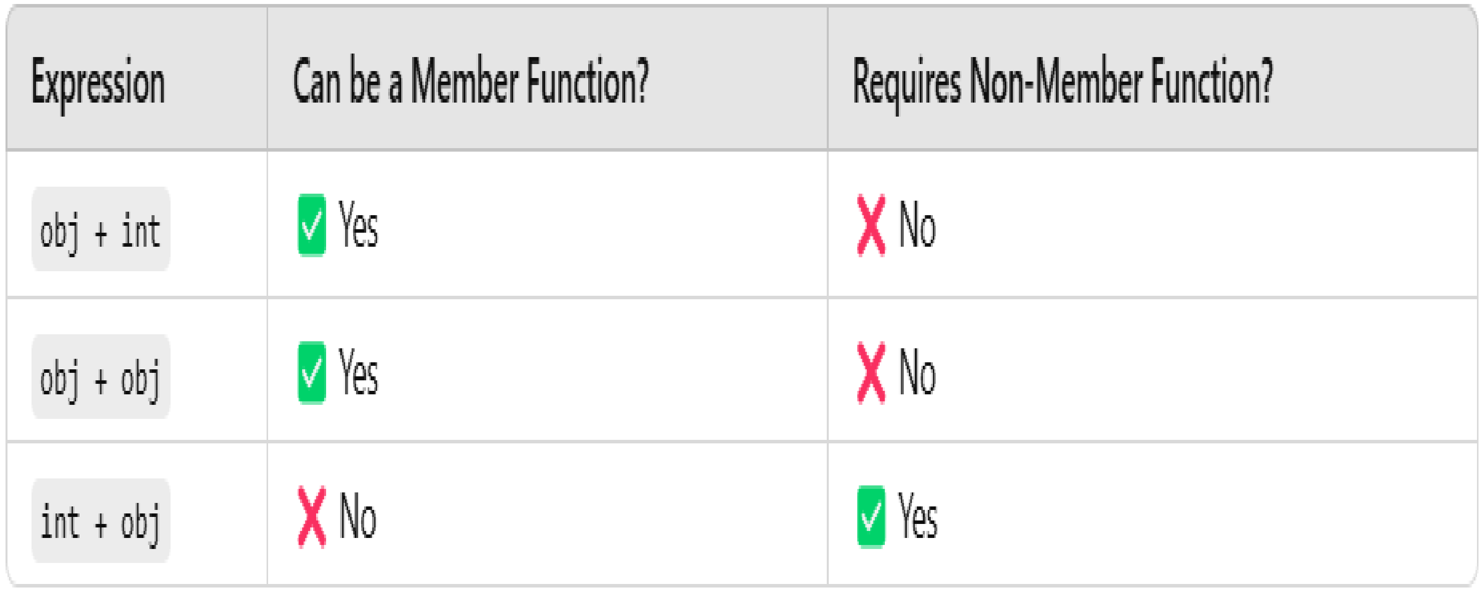
Non-Member Function (Left Operand is Not an Object of the Class)

```
#include <iostream>
using namespace std;
class Number {
    int value;
public:
    Number(int v) : value(v) {}
    // Friend function to overload + when left operand is int
    friend Number operator+(int lhs, const Number& obj);
    void display() { cout << "Value: " << value << endl; }};
// Overloading + when left operand is an int
Number operator+(int lhs, const Number& obj) {
    return Number(lhs + obj.value);}
```

```
int main() {
   Number n(30);
   Number result = 10 + n;  // Left operand is int, not a Number object
//operator+(10, obj);  Equivalent to 10+n
   result.display();
   return 0;}
```

Why Does obj + int Work in a Class Member Function?

```
class Number {
   int value;
public:
   Number(int v) : value(v) {}
Number operator+(int rhs) {
      return Number(value + rhs);
   }};
int main() {
   Number n1(10);
   Number result = n1 + 5; // Works fine
//n1.operator+(5).
}
```



Expression	Can be a Member Function?	Requires Non-Member Function?
`obj + int`	✅ Yes	❌ No
`obj + obj`	✅ Yes	❌ No
`int + obj`	❌ No	✅ Yes

Task

Calculate Area of Triangle the one class is acute angle(less than 90 degree) and another one is obtuse angle (greater than 90 degree) both class wants to access only one function which is “Area of Triangle” which is independent of all classes.