UNIT-2

Date: 1/4/2021 ***Chapter-2***

Reetu Bhardwaj

Polymorphism

The term "Polymorphism" is the combination of

"poly" + "morphs" which means many forms. It is a greek word**.**

That is, the same entity (function or operator) behaves differently in different scenarios. For example,

The + operator in C++ is used to perform two specific functions. When it is used with numbers (integers and floating-point numbers), it performs addition.

Example just think about the word “Right”

It can be used in 3 different contextual statement

Right

1. **Let me know the direction, its left or right?**
2. **Fight for your right**
3. **Tell me, am I wrong or Right?**

Another example:

```
int a = 5;

int b = 6;

int sum = a + b;    // sum = 11
```

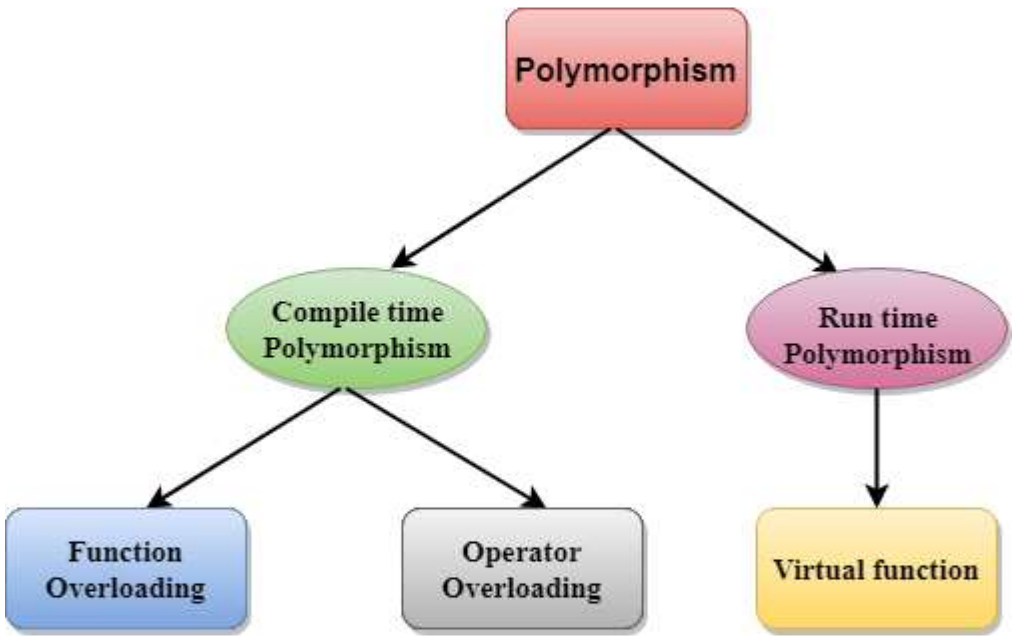
And when we use the + operator with strings, it performs string concatenation. For example,

```
string firstName = "abc ";

string lastName = "xyz";

string name = firstName + lastName;

// name = "abc xyz"
```

Types of Polymorphism

In C++ polymorphism is mainly divided into two types:

- Compile time Polymorphism
- Runtime Polymorphism



Compile Time Polymorphism:

The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time.

The compiler matches the function call with the correct function definition at compile time. It is also known as **Static Binding or Early Binding.** By default, the compiler goes to the function definition which has been called during compile time.

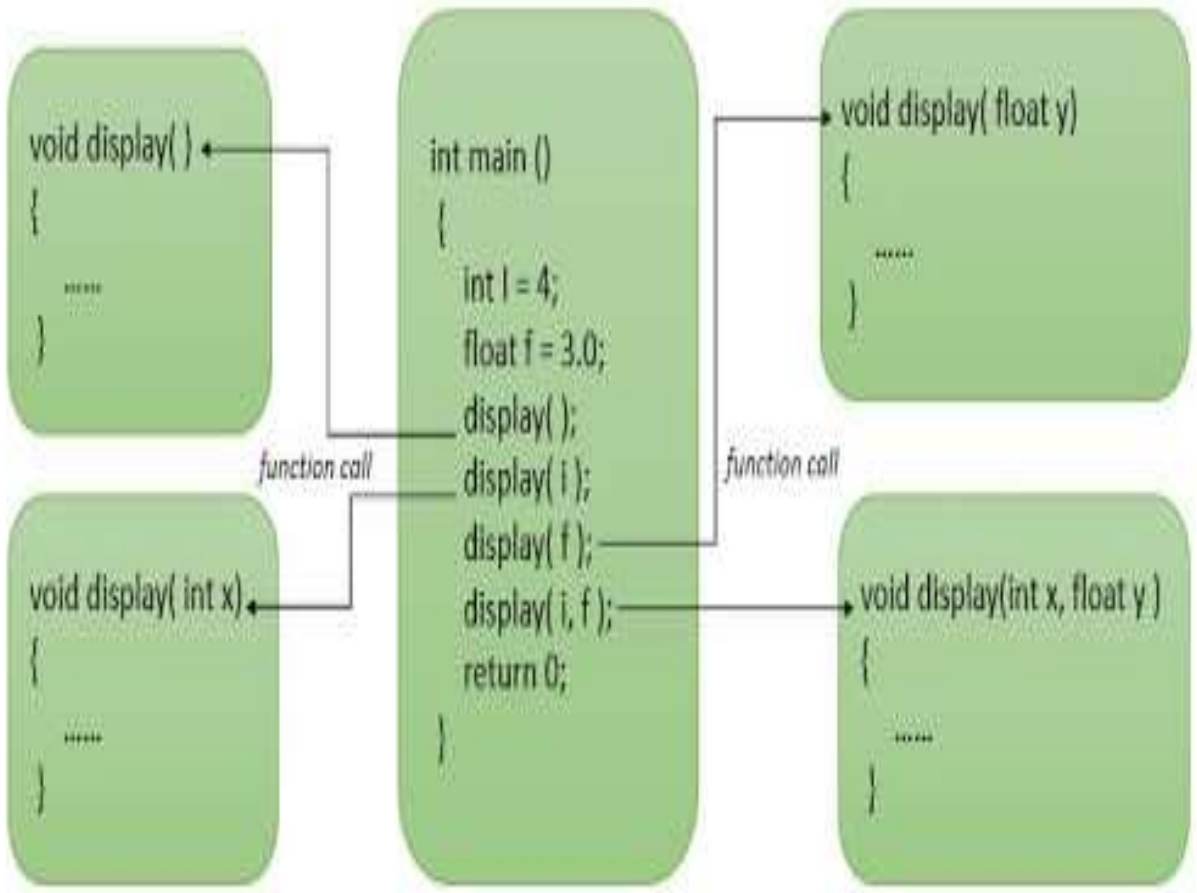
This type of polymorphism is achieved by function overloading or operator overloading

Function Overloading:

When there are multiple functions with same name but different parameters then these functions are said to be overloaded.

Functions can be overloaded by change in number of arguments or/and change in type of arguments.

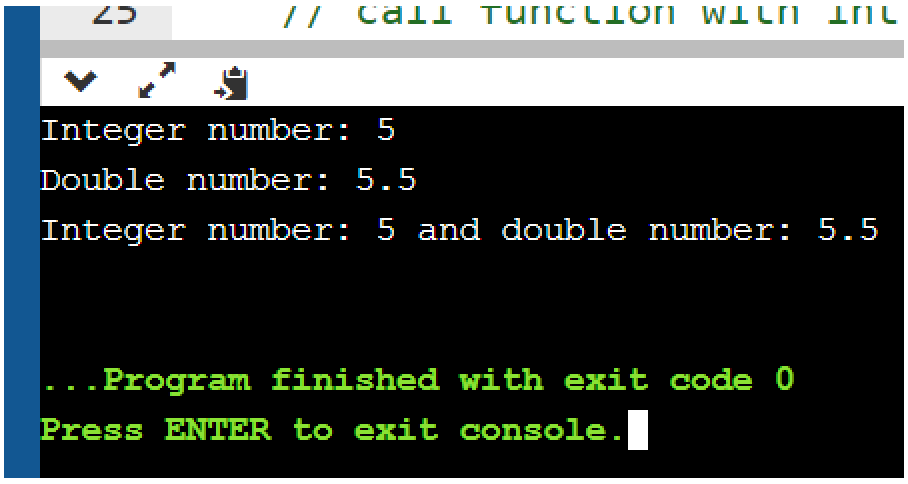
}



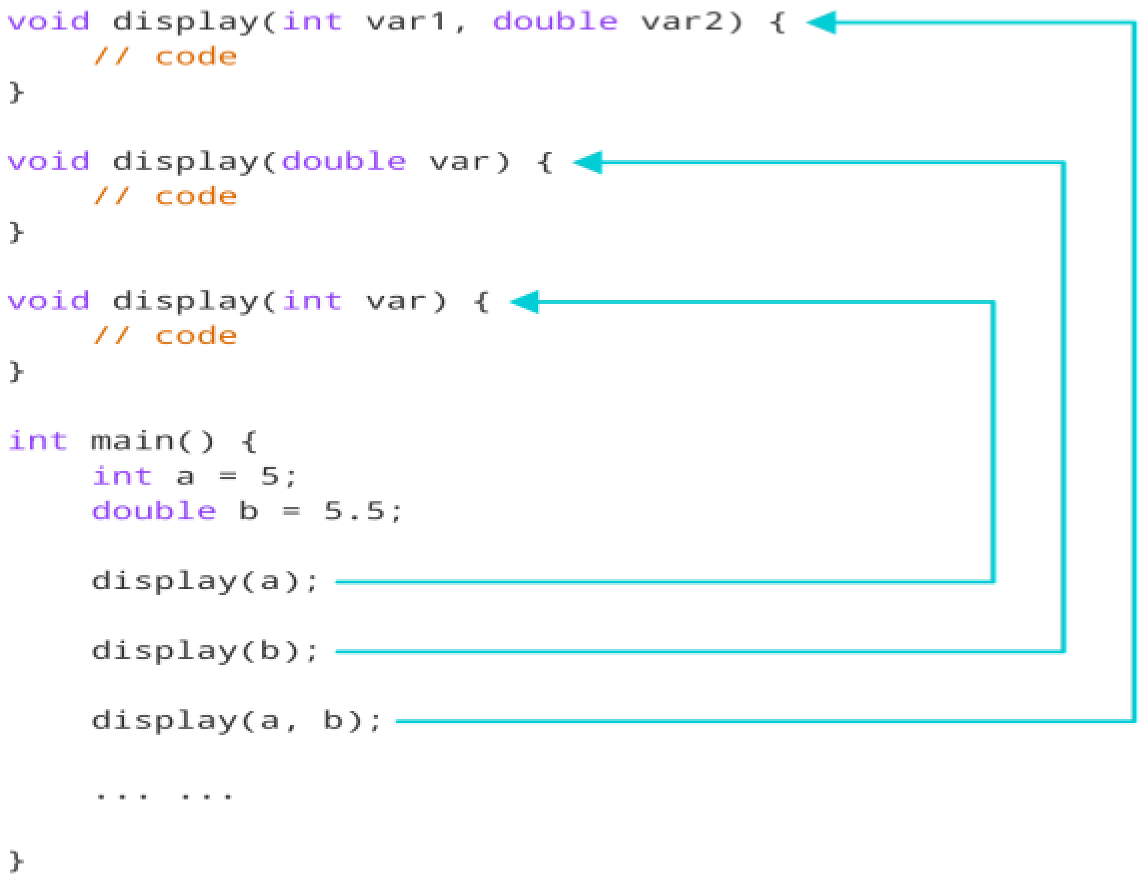
Example:

```
#include <iostream>
using namespace std;
// function with 2 parameters
void display(int var1, int var2) {
    cout << "Integer number: " << var1;
    cout << " and double number: " << var2 << endl;
}
```

```
// function with double type single parameter
void display(double var) {
    cout << "Double number: " << var << endl;
}
// function with int type single parameter
void display(int var) {
    cout << "Integer number: " << var << endl;
}
int main() {
   int a = 5;
    double b = 5.5;
 // call function with int type parameter
    display(a);
 // call function with double type parameter
    display(b);
// call function with 2 parameters
    display(a, b);
 return 0;
```



```
Integer number: 5
Double number: 5.5
Integer number: 5 and double number: 5.5


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



```
void display(int var1, double var2) {
    // code
}

void display(double var) {
    // code
}

void display(int var) {
    // code
}

int main() {
    int a = 5;
    double b = 5.5;

    display(a);

    display(b);

    display(a, b);

    ... ...

}
```

<u>Advantages of function Over loading in C++</u>

- ❖ We use function overloading to save the memory space, consistency, and readability of our program.

- ❖ With the use function overloading concept, we can develop more than one function with the same name

- ❖ Function overloading shows the behavior of polymorphism that allows us to get different behavior, although there will be some link using the same name of the function.

- ❖ Function overloading speeds up the execution of the program.

- ❖ Function overloading is used for code reusability and also to save memory.

- ❖ Code maintenance is easy.

Function Overloading and Ambiguity

When the compiler is unable to decide which function it should invoke first among the overloaded functions, this situation is known as function overloading ambiguity.

The compiler does not run the program if it shows ambiguity error.

Causes of Function Overloading ambiguity:

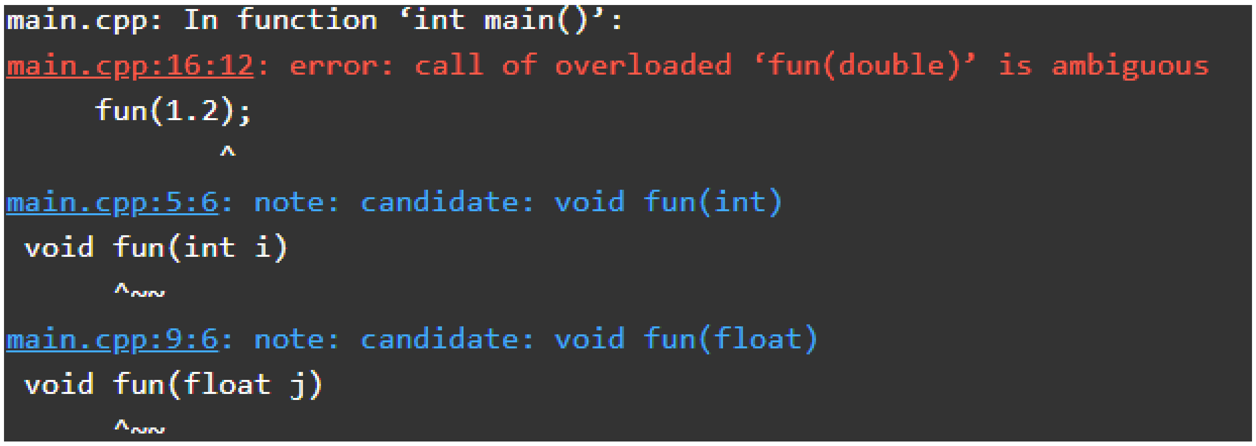
- ❖ Type Conversion.
- ❖ Function with default arguments.
- ❖ Function with a pass by reference

1. Ambiguity due to Type Conversion

Example:

```
#include<iostream>

using namespace std;

void fun(int);

void fun(float);

void fun(int i)

{

   std::cout << "Value of i is : " <<i<< std::endl;

}
```

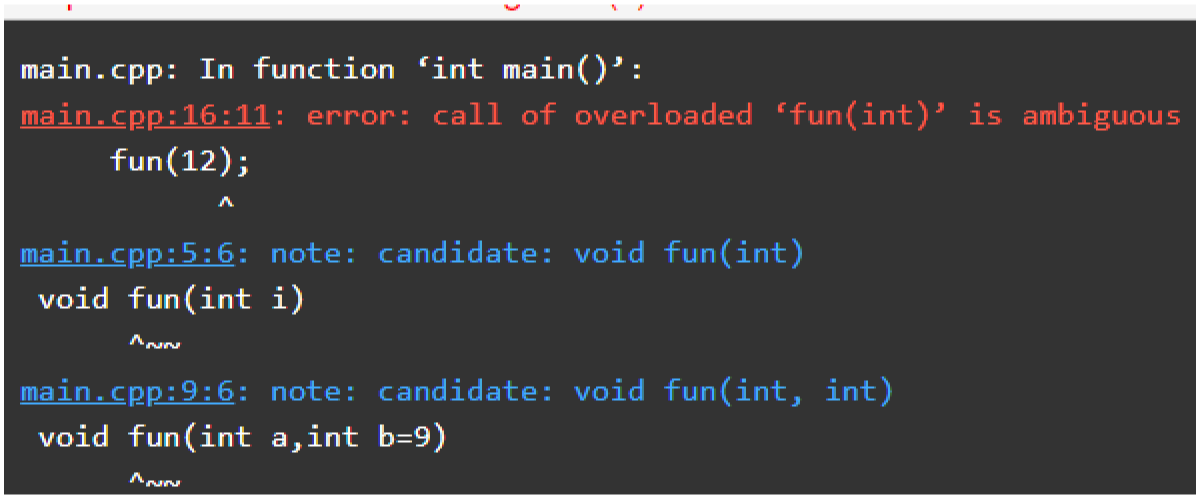
```
void fun(float j)

{

    std::cout << "Value of j is : " <<j<< std::endl;

}

int main()

{

    fun(12);

    fun(1.2);

    return 0;

}
```



```
main.cpp: In function 'int main()':
main.cpp:16:12: error: call of overloaded 'fun(double)' is ambiguous
     fun(1.2);
            ^
main.cpp:5:6: note: candidate: void fun(int)
 void fun(int i)
      ^~~
main.cpp:9:6: note: candidate: void fun(float)
 void fun(float j)
      ^~~
```

2. Ambiguity due to Function with default arguments.

```
#include<iostream>

using namespace std;

void fun(int);

void fun(int,int);

void fun(int i)

{

   std::cout << "Value of i is : " <<i<< std::endl;

}

void fun(int a,int b=9)

{

   std::cout << "Value of a is : " <<a<< std::endl;

   std::cout << "Value of b is : " <<b<< std::endl;

}

int main()

{

   fun(12);

   return 0;

}
```



```
main.cpp: In function ‘int main()’:
main.cpp:16:11: error: call of overloaded ‘fun(int)’ is ambiguous
     fun(12);
           ^
main.cpp:5:6: note: candidate: void fun(int)
 void fun(int i)
      ^~~
main.cpp:9:6: note: candidate: void fun(int, int)
 void fun(int a,int b=9)
      ^~~
```

The fun(int a, int b=9) can be called in two ways: first is by calling the function with one argument, i.e., fun(12) and another way is calling the function with two arguments, i.e., fun(4,5).

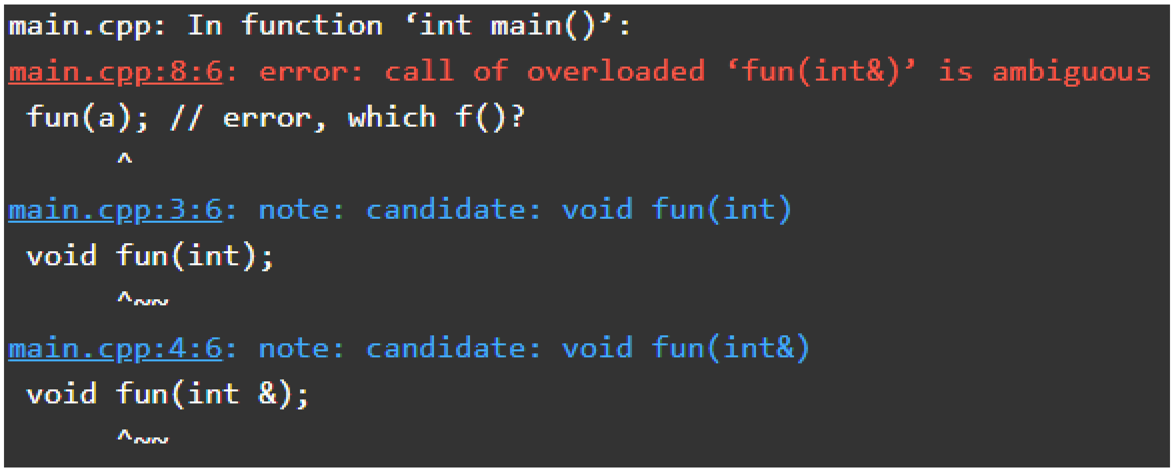
The fun(int i) function is invoked with one argument. Therefore, the compiler could not be able to select among fun(int i) and fun(int a,int b=9).

3. Ambiguity due to Function with a pass by reference

Example

```
#include <iostream>
using namespace std;
void fun(int);
void fun(int &);
int main()
{
int a=10;
fun(a); // error, which f()?
return 0;
}
void fun(int x)
{
std::cout << "Value of x is : " <<x<< std::endl;
}
void fun(int &b)
{
std::cout << "Value of b is : " <<b<< std::endl;
```

}



```
main.cpp: In function ‘int main()’:
main.cpp:8:6: error: call of overloaded ‘fun(int&)’ is ambiguous
 fun(a); // error, which f()?
      ^
main.cpp:3:6: note: candidate: void fun(int)
 void fun(int);
      ^~~
main.cpp:4:6: note: candidate: void fun(int&)
 void fun(int &);
      ^~~
```

The first function takes one integer argument and the second function takes a reference parameter as an argument. In this case, the compiler does not know which function is needed by the user as there is no syntactical difference between the fun(int) and fun(int &).

Differences between function overloading and function overriding

1) Function Overloading happens in the same class when we declare same functions with different arguments in the same class. Function Overriding is happens in the child class when child class overrides parent class function.

2) In function overloading function signature should be different for all the overloaded functions. In function overriding the signature of both the functions (overriding function and overridden function) should be same.

3) Overloading happens at the compile time thats why it is also known as compile time polymorphism while overriding happens at run time which is why it is known as run time polymorphism.

4) In function overloading we can have any number of overloaded functions. In function overriding we can have only one overriding function in the child class.

What is Operator Overloading?

Using operator overloading in C++, you can specify more than one meaning for an operator in one scope.

- ❖ The purpose of operator overloading is to provide a special meaning of an operator for a user-defined data type.
- ❖ With the help of operator overloading, you can redefine the majority of the C++ operators.

Note: We cannot use operator overloading for fundamental data types like int, float, char and so on.

Syntax for C++ Operator Overloading

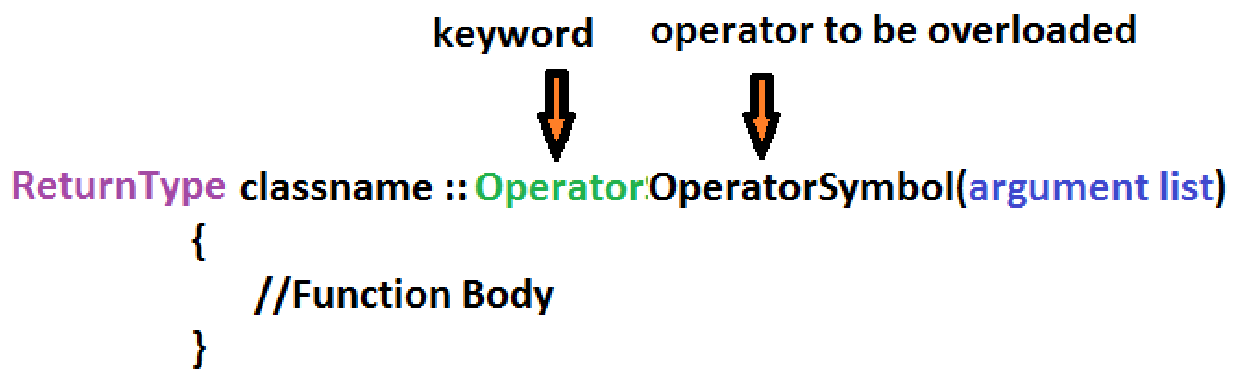
To overload an operator, we use a special operator function. We define the function inside the class's objects/variables we want the overloaded operator to work with.

```
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 (When operator function is defined outside the class)



```
ReturnType classname :: Operator OperatorSymbol(argument list)
        {
            //Function Body
        }
```

What is the difference between operator functions and normal functions?

Operator functions are same as normal functions. The only differences are, name of an operator function is always operator keyword followed by symbol of operator and operator functions are called when the corresponding operator is used.

Operator that cannot be overloaded are as follows:

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

Rules for the operator overloading.

- Only built-in operators can be overloaded. If some operators are not present in C++, we cannot overload them.
- The basic meaning of operator can’t be changed
- The precedence of the operators remains same.
- We cannot overload operators for built-in data types. At least one user defined data types must be there.
- Object is being passed as argument
- Operator (function)should be member of same class, so we can pass object as an argument
- Some operators like assignment “=”, address “&” and comma “,” are by default overloaded.

Operator Overloading can be done by using three approaches, they are

1. **Overloading unary operator.**
2. **Overloading binary operator.**
3. **Overloading binary operator using a friend function.**

1. Unary operator Overloading

Unlike the operators you've seen so far, the positive (+), negative (-) ,logical not (!), increment (++), and decrement (--) these all operators are unary operators, which means they only operate on one operand.
Because they only operate on the object they are applied to, typically unary operator overloads are implemented as member functions.

Note: when we are overloading the post-fix operator i.e Post increment(++) and post decrement(--), we have to pass int as dummy argument to the operator function*

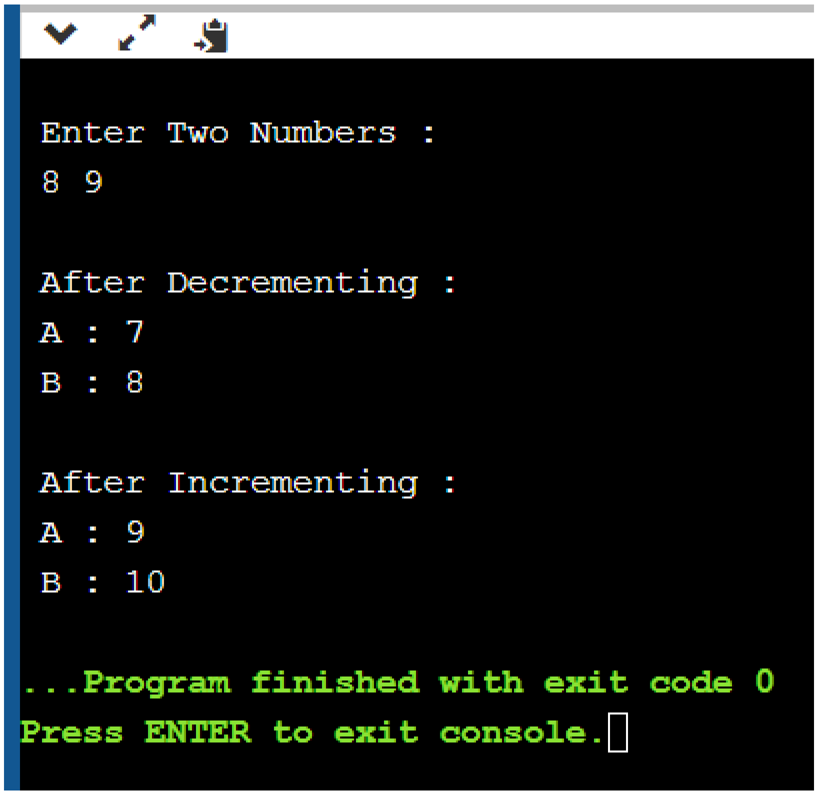
And in calling overloaded operator will be used in right side of operand
e.g obj++

Example:

```
#include<iostream>

using namespace std;

class IncreDecre

{

        int a, b;   //data members
```

```
public:
  void accept() //member function
  {
        cout<<"\n Enter Two Numbers : \n";
        cout<<" ";
        cin>>a;
        cout<<" ";
        cin>>b;
  }
  void operator--() //Overload Unary Decrement
  {
        a--;
        b--;
  }
  void operator++() //Overload Unary Increment
  {
        a++;
        b++;
  }
  void display()
  {
```

```
		cout<<"\n A : "<<a;
		cout<<"\n B : "<<b;
	}
};
int main()
{
	IncreDecre id;
	id.accept();
	--id;
	cout<<"\n After Decrementing : ";
	id.display();
	++id;
	++id;
	cout<<"\n\n After Incrementing : ";
	id.display();
	return 0;
}
```

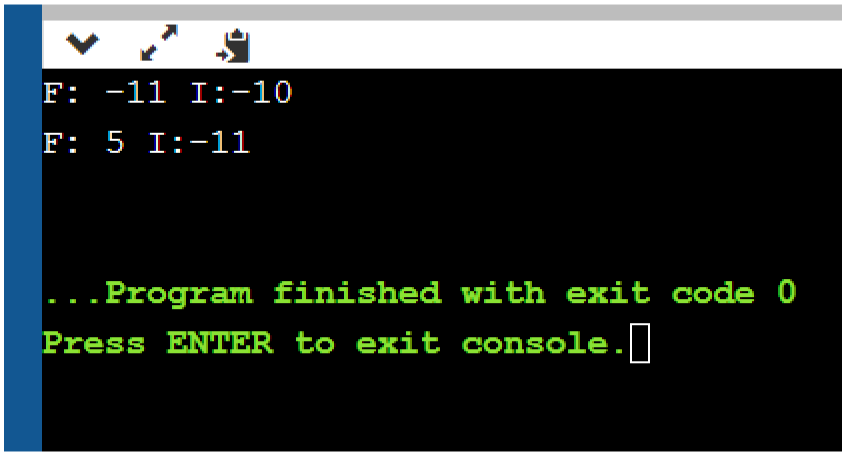


```
Enter Two Numbers :
8 9

After Decrementing :
A : 7
B : 8

After Incrementing :
A : 9
B : 10

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

Example 2:

```
#include <iostream>
using namespace std;
 class Distance {
   private:
      int feet;
      int inches;
      public:  // required constructors
        Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }
      void displayDistance() {
         cout << "F: " << feet << " I:" << inches <<endl;
      }
// overloaded minus (-) operator
      Distance operator- () {
         feet = -feet;
         inches = -inches;
         return Distance(feet, inches);
      }
};
int main() {
   Distance D1(11, 10), D2(-5, 11);
   -D1;
   D1.displayDistance();
  -D2;
   D2.displayDistance();
```

```
 return 0;
}
```



```
F: -11 I:-10
F: 5 I:-11


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

2. Overloading of Binary Operator:

In binary operator overloading function, there should be one argument to be passed.

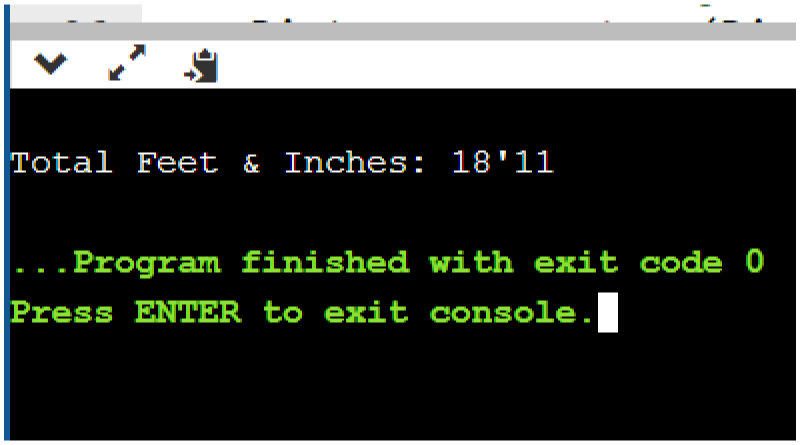
It is overloading of an operator operating on two operands.

Example 1:

```
#include <iostream>
 using namespace std;
  class Distance {
public:
    // Member Object
    int feet, inch;
```

```
   // No Parameter Constructor
   Distance()
   {
       feet = 0;
       inch = 0;
   }
   // Constructor to initialize the object's value
   // Parametrized Constructor
Distance(int f, int i)
   {
       feet = f;
       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 = feet + d2.feet;
```

```
        d3.inch = inch + d2.inch;
    // Return the resulting object
        return d3;
     }
};
  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; }
```

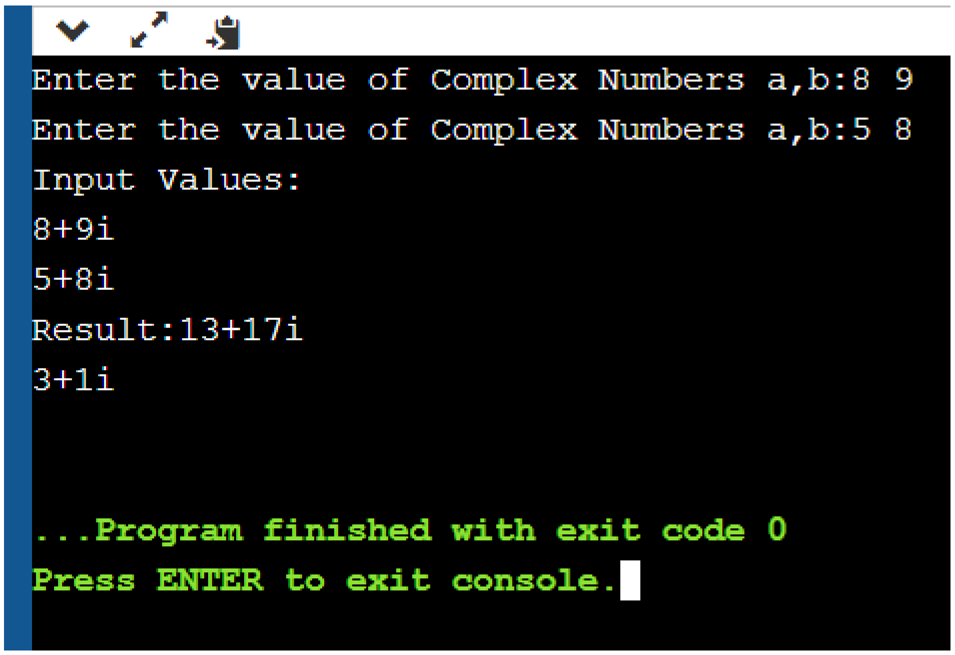


```
Total Feet & Inches: 18'11

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

Example 2:

```
#include<iostream>
using namespace std;
class complex
{
int a, b;
public:
void get_data()
{
cout << "Enter the value of Complex Numbers a,b:";
cin >> a>>b;
}
complex operator+(complex ob) // overaloded operator function +
{
complex t;
t.a = a + ob.a;
t.b = b + ob.b;
return (t);
}
complex operator-(complex ob)// overaloded operator function -
{
complex t;
t.a = a - ob.a;
t.b = b - ob.b;
return (t);
}
void display()
{
cout << a << "+" << b << "i" << "\n";
}
};
Int main()
{
complex obj1, obj2, result, result1;
obj1.get_data();
obj2.get_data();
```

```
result = obj1 + obj2;

result1 = obj1 - obj2;

cout << "Input Values:\n";

obj1.display();

obj2.display();

cout << "Result:";

result.display();

result1.display();

}
```



```
Enter the value of Complex Numbers a,b:8 9
Enter the value of Complex Numbers a,b:5 8
Input Values:
8+9i
5+8i
Result:13+17i
3+1i


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

3. Operator Overloading Using friend function

Friend function using operator overloading offers better flexibility to the class.

- When you overload a unary operator you have to pass one argument.
- When you overload a binary operator you have to pass two arguments.
- Friend function can access private members of a class directly.
- Friend-function is non -member function of class

Syntax:

friend return-type operator operator-symbol (Variable 1, Varibale2)

{

//Statements;

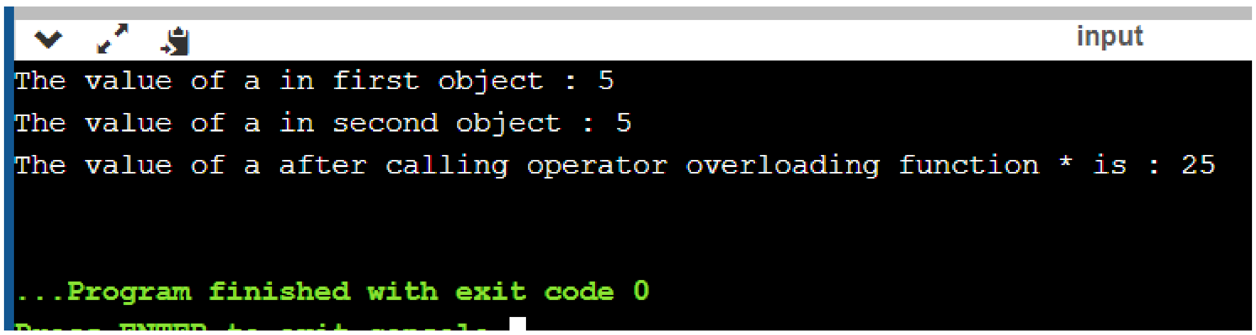
}

Example 1 (Binary Operator Overloading Using Friend Function(Two arguments)

```
#include<iostream>

using namespace std;

class A

{

private:

int a;
```

```
public:

void set_a();

void get_a();

friend A operator *(A,A);      //Binary operator * overloaded friend
function,

 };

//Definition of set_a() function

void A :: set_a()

{

a = 5;

}

//Definition of get_a() function

void A :: get_a()

{

cout<< a <<"\n";

}

//Definition of overloaded binary operator * friend function

A operator *(A ob1, A ob2)

{

A temp;
```

```
temp.a = ob1.a * ob2.a;
return temp;
}
int main()
{
A ob1, ob2;
ob1.set_a();
ob2.set_a();
cout<<"The value of a in first object : ";
ob1.get_a();
cout<<"The value of a in second object : ";
ob2.get_a();
A ob3 = ob1 * ob2;  //calling overloaded function
cout<<"The value of a after calling operator overloading function * is :
";
ob3.get_a();
}
```

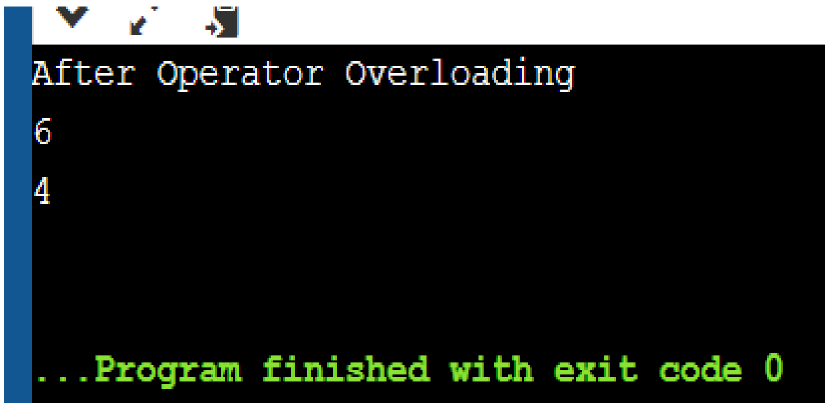


```
The value of a in first object : 5
The value of a in second object : 5
The value of a after calling operator overloading function * is : 25


...Program finished with exit code 0
```

Example 2: Unary Operator Overloading by using Friend Function (Using one Argument)

```
#include<iostream>
using namespace std;
class increment
{
private:
      int a,b;
public:
increment()
{    }
  increment(int p,int q) //parameterized constructor
     {     a=p;
           b=q;   }
      void display()
      {
       cout<<a <<endl <<b<<endl;
    }
   friend increment operator ++ (increment obj);
};
```

```
increment operator ++ (increment obj)
{
        increment t;
        t.a=++(obj.a);
        t.b=++(obj.b);
        return t;
}
int main(){
increment x(5,3),y;
 y=++x;
 cout<<"After Operator Overloading \n";
 y.display(); }
```



```
After Operator Overloading
6
4


...Program finished with exit code 0
```

Constructor Overloading???

****We have already discussed this topic in Unit 1, so kindly refer unit 1's notes****