CL-217 OBJECT ORIENTED PROGRAMMING LAB

LAB Manual 8

INSTRUCTOR: MUHAMMAD HAMZA

SEMESTER SPRING 2020

C++ Function Overloading

Two or more functions having same name but different argument(s) are known as overloaded functions. In this article, you will learn about function overloading with examples.

Function refers to a segment that groups code to perform a specific task.

In C++ programming, two functions can have same name if number and/or type of arguments passed are different.

These functions having different number or type (or both) of parameters are known as overloaded functions. For example:

```
int test() { }

int test(int a) { }

float test(double a) { }
int test(int a, double b) { }
```

Here, all 4 functions are overloaded functions because argument(s) passed to these functions are different.

Notice that, the return type of all these 4 functions are not same. Overloaded functions may or may not have different return type but it should have different argument(s).

```
// Error code

int test(int a) { }

double test(int b){ }
```

The number and type of arguments passed to these two functions are same even though the return type is different. Hence, the compiler will throw error.

Example 1: Function Overloading

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

void display(int);
void display(float);
void display(int, float);

int main() {

    int a = 5;
    float b = 5.5;

    display(a);
    display(b);
    display(a, b);

    return 0;
}

void display(int var) {
    cout << "Integer number: " << var << endl;
```

```
}


void display(float var) {
   cout << "Float number: " << var << endl;
}


void display(int var1, float var2) {
   cout << "Integer number: " << var1;
   cout << " and float number:" << var2;
}
```

Output

```
Integer number: 5
Float number: 5.5
Integer number: 5 and float number: 5.5
```

Here, the `display()` function is called three times with different type or number of arguments.

The return type of all these functions are same but it's not necessary.

Example 2: Function Overloading

```
// Program to compute absolute value
// Works both for integer and float

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

```
int absolute(int);
float absolute(float);

int main() {
   int a = -5;
   float b = 5.5;

   cout << "Absolute value of " << a << " = " << absolute(a) << endl;
   cout << "Absolute value of " << b << " = " << absolute(b);
   return 0;
}

int absolute(int var) {
    if (var < 0)
        var = -var;
   return var;
}

float absolute(float var){
   if (var < 0.0)
       var = -var;
   return var;
}
```

Output

```
Absolute value of -5 = 5
Absolute value of 5.5 = 5.5
```

In the above example, two functions `absolute()` are overloaded.

Both functions take single argument. However, one function takes integer as an argument and other takes float as an argument.

When `absolute()` function is called with integer as an argument, this function is called:

```
int absolute(int var) {
    if (var < 0)
        var = -var;
    return var;
}
```

When `absolute()` function is called with float as an argument, this function is called:

```
float absolute(float var){
    if (var < 0.0)
        var = -var;
    return var;
}
```

Operators Overloading in C++

You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well.

Overloaded operators are functions with special names: the keyword "operator" followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

```
Box operator+(const Box&);
```

declares the addition operator that can be used to add two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions. In case we define above function as non-member function of a class then we would have to pass two arguments for each operand as follows −

```
Box operator+(const Box&, const Box&);
```

Following is the example to show the concept of operator over loading using a member function. Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using this operator as explained below −

Live Demo

```
#include <iostream>

using namespace std;


class Box {

   public:

      double getVolume(void) {

         return length * breadth * height;

      }

      void setLength( double len ) {

         length = len;

      }

      void setBreadth( double bre ) {
```

```
      breadth = bre;
   }
   void setHeight( double hei ) {
      height = hei;
   }

   // Overload + operator to add two Box objects.
   Box operator+(const Box& b) {
      Box box;
      box.length = this->length + b.length;
      box.breadth = this->breadth + b.breadth;
      box.height = this->height + b.height;
      return box;
   }

  private:
   double length;      // Length of a box
   double breadth;     // Breadth of a box
   double height;      // Height of a box
};

// Main function for the program
int main() {
  Box Box1;                // Declare Box1 of type Box
  Box Box2;                // Declare Box2 of type Box
  Box Box3;                // Declare Box3 of type Box
```

```
  double volume = 0.0;     // Store the volume of a box here

  // box 1 specification
  Box1.setLength(6.0);
  Box1.setBreadth(7.0);
  Box1.setHeight(5.0);

  // box 2 specification
  Box2.setLength(12.0);
  Box2.setBreadth(13.0);
  Box2.setHeight(10.0);

  // volume of box 1
  volume = Box1.getVolume();
  cout << "Volume of Box1 : " << volume <<endl;

  // volume of box 2
  volume = Box2.getVolume();
  cout << "Volume of Box2 : " << volume <<endl;

  // Add two object as follows:
  Box3 = Box1 + Box2;

  // volume of box 3
  volume = Box3.getVolume();
  cout << "Volume of Box3 : " << volume <<endl;
```

```
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

Overloadable/Non-overloadableOperators

Following is the list of operators which can be overloaded −

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

Following is the list of operators, which can not be overloaded −

::	.*	.	?:

Overloading an Operator: Some Restrictions

When overloading an operator, keep the following in mind:

1. You cannot change the precedence of an operator.

2. The associativity cannot be changed. (For example, the associativity of the arithmetic operator addition is from left to right, and it cannot be changed.)

3. Default parameters cannot be used with an overloaded operator.

4. You cannot change the number of parameters an operator takes.

5. You cannot create new operators. Only existing operators can be overloaded.

6. The operators that cannot be overloaded are:

.

.*

::

?:

sizeof

7. The meaning of how an operator works with built-in types, such as int , remains the same. That is, you cannot redefine how operators work with built-in data types.

8. Operators can be overloaded either for objects of the user-defined

types, or for a combination of objects of the user-defined type and

objects of the built-in type.

Unary Operators Overloading in C++:

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 logical not (!) 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 sometimes they can be used as postfix as well like obj++ or obj--.

Following example explain how minus (-) operator can be overloaded for prefix as well as postfix usage.

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

class Distance {
  private:
    int feet;             // 0 to infinite
    int inches;           // 0 to 12

  public:
    // required constructors
    Distance() {
       feet = 0;
       inches = 0;
    }
    Distance(int f, int i) {
       feet = f;
       inches = i;
    }

    // method to display distance
    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;                    // apply negation
  D1.displayDistance();   // display D1

  -D2;                    // apply negation
  D2.displayDistance();   // display D2

  return 0;
}
```

When the above code is compiled and executed, it produces the following result −

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

Hope above example makes your concept clear and you can apply similar concept to overload Logical Not Operators (!).

Binary Operators Overloading in C++:

The binary operators take two arguments and following are the examples of Binary operators. You use binary operators very frequently like addition (+) operator, subtraction (-) operator and division (/) operator.

Following example explains how addition (+) operator can be overloaded. Similar way, you can overload subtraction (-) and division (/) operators.

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

class Box {
   double length;      // Length of a box
   double breadth;     // Breadth of a box
   double height;      // Height of a box

   public:

   double getVolume(void) {
      return length * breadth * height;
   }

   void setLength( double len ) {
      length = len;
   }

   void setBreadth( double bre ) {
      breadth = bre;
   }

   void setHeight( double hei ) {
      height = hei;
   }

   // Overload + operator to add two Box objects.
   Box operator+(const Box& b) {
      Box box;
      box.length = this->length + b.length;
      box.breadth = this->breadth + b.breadth;
      box.height = this->height + b.height;
      return box;
   }
};
```

```
// Main function for the program
int main() {
   Box Box1;                // Declare Box1 of type Box
   Box Box2;                // Declare Box2 of type Box
   Box Box3;                // Declare Box3 of type Box
   double volume = 0.0;     // Store the volume of a box here

   // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);

   // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;

   // Add two object as follows:
   Box3 = Box1 + Box2;

   // volume of box 3
   volume = Box3.getVolume();
   cout << "Volume of Box3 : " << volume <<endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

Relational Operators Overloading in C++:

There are various relational operators supported by C++ language like (<, >, <=, >=, ==, etc.) which can be used to compare C++ built-in data types.

You can overload any of these operators, which can be used to compare the objects of a class.

Following example explains how a < operator can be overloaded and similar way you can overload other relational operators.

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

class Distance {
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12

   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }

      // method to display distance
      void displayDistance() {
         cout << "F: " << feet << " I:" << inches <<endl;
      }

      // overloaded minus (-) operator
      Distance operator- () {
         feet = -feet;
         inches = -inches;
         return Distance(feet, inches);
      }

      // overloaded < operator
```

```
    bool operator <(const Distance& d) {
       if(feet < d.feet) {
          return true;
       }
       if(feet == d.feet && inches < d.inches) {
          return true;
       }

       return false;
    }
};

int main() {
  Distance D1(11, 10), D2(5, 11);

  if( D1 < D2 ) {
    cout << "D1 is less than D2 " << endl;
  } else {
    cout << "D2 is less than D1 " << endl;
  }

  return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
D2 is less than D1
```

If an operator function is a member function of a class, then the leftmost operand of that operator must be an object of that class. Therefore, the operator function that overloads the insertion operator, << , or the extraction operator, >> , for a class must be a nonmember function of that class.

Input/Output Operators (>> / <<)Overloading in C++:

If an operator function is a member function of a class, then the leftmost operand of that operator must be an object of that class. Therefore, the operator function that overloads the insertion operator, << , or the extraction operator, >> , for a class must be a nonmember function of that class.

C++ is able to input and output the built-in data types using the stream extraction operator >> and the stream insertion operator <<. The stream insertion and stream extraction operators also can be overloaded to perform input and output for user-defined types like an object.

Here, it is important to make the operator overloading function as a friend of the class because it would be called without creating an object.

Following example explains how extraction operator >> and insertion operator <<.

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

class Distance {
  private:
    int feet;             // 0 to infinite
    int inches;           // 0 to 12

  public:
    // required constructors
    Distance() {
      feet = 0;
      inches = 0;
    }
    Distance(int f, int i) {
      feet = f;
      inches = i;
    }
    friend ostream &operator<<( ostream &output, const Distance &D ) {
      output << "F : " << D.feet << " I : " << D.inches;
      return output;
    }
```

```
    friend istream &operator>>( istream  &input, Distance &D ) {
       input >> D.feet >> D.inches;
       return input;
    }
};

int main() {
  Distance D1(11, 10), D2(5, 11), D3;

  cout << "Enter the value of object : " << endl;
  cin >> D3;
  cout << "First Distance : " << D1 << endl;
  cout << "Second Distance :" << D2 << endl;
  cout << "Third Distance :" << D3 << endl;

  return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
$./a.out
Enter the value of object :
70
10
First Distance : F : 11 I : 10
Second Distance :F : 5 I : 11
Third Distance :F : 70 I : 10
```

Why we are returning stream object from the function ?

Find answer in DS Malik Book Page : 917

Overloading Increment ++ & Decrement -- :

The increment (++) and decrement (--) operators are two important unary operators available in C++.

Following example explain how increment (++) operator can be overloaded for prefix as well as postfix usage. Similar way, you can overload operator (--).

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

```
class Time {
  private:
    int hours;             // 0 to 23
    int minutes;           // 0 to 59

  public:
    // required constructors
    Time() {
       hours = 0;
       minutes = 0;
    }
    Time(int h, int m) {
       hours = h;
       minutes = m;
    }

    // method to display time
    void displayTime() {
       cout << "H: " << hours << " M:" << minutes <<endl;
    }

    // overloaded prefix ++ operator
    Time operator++ () {
       ++minutes;          // increment this object
       if(minutes >= 60) {
          ++hours;
          minutes -= 60;
       }
       return Time(hours, minutes);
    }

    // overloaded postfix ++ operator
    Time operator++( int ) {

       // save the orignal value
       Time T(hours, minutes);

       // increment this object
       ++minutes;

       if(minutes >= 60) {
          ++hours;
          minutes -= 60;
       }
```

```
      // return old original value
      return T;
   }
};

int main() {
  Time T1(11, 59), T2(10,40);

  ++T1;                  // increment T1
  T1.displayTime();      // display T1
  ++T1;                  // increment T1 again
  T1.displayTime();      // display T1

  T2++;                  // increment T2
  T2.displayTime();      // display T2
  T2++;                  // increment T2 again
  T2.displayTime();      // display T2
  return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
H: 12 M:0
H: 12 M:1
H: 10 M:41
H: 10 M:42
```

Assignment Operators Overloading in C++

You can overload the assignment operator (=) just as you can other operators and it can be used to create an object just like the copy constructor.

Following example explains how an assignment operator can be overloaded.

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

class Distance {
  private:
    int feet;             // 0 to infinite
    int inches;           // 0 to 12
```

```
   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }
      void operator = (const Distance &D ) {
         feet = D.feet;
         inches = D.inches;
      }

      // method to display distance
      void displayDistance() {
         cout << "F: " << feet <<  " I:" <<  inches << endl;
      }
};

int main() {
   Distance D1(11, 10), D2(5, 11);

   cout << "First Distance : ";
   D1.displayDistance();
   cout << "Second Distance :";
   D2.displayDistance();

   // use assignment operator
   D1 = D2;
   cout << "First Distance :";
   D1.displayDistance();

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
First Distance : F: 11 I:10
Second Distance :F: 5 I:11
First Distance :F: 5 I:11
```

What is the difference between copy constructor and = operator overload? (FIND OUT)

Overloading the subscript operator:

When working with arrays, we typically use the subscript operator ([]) to index specific elements of an array:

```
myArray[0] = 7; // put the value 7 in the first element of the array
```

However, consider the following IntList class, which has a member variable that is an array:

```
class IntList
{
private:
    int m_list[10]{};
};

int main()
{
    IntList list{};
    // how do we access elements from m_list?
    return 0;
}
```

Because the m_list member variable is private, we can not access it directly from variable list. This means we have no way to directly get or set values in the m_list array. So how do we get or put elements into our list?

Without operator overloading, the typical method would be to create access functions:

```
class IntList
{
private:
    int m_list[10]{};

public:
    void setItem(int index, int value) { m_list[index] = value; }
    int getItem(int index) const { return m_list[index]; }
};
```

While this works, it's not particularly user friendly. Consider the following example:

```
int main()
{
    IntList list{};
    list.setItem(2, 3);

    return 0;
}
```

Are we setting element 2 to the value 3, or element 3 to the value 2? Without seeing the definition of setItem(), it's simply not clear.

You could also just return the entire list and use operator[] to access the element:

```
class IntList
{
private:
    int m_list[10]{};

public:
    int* getList() { return m_list; }
};
```

While this also works, it's syntactically odd:

```
int main()
{
    IntList list{};
    list.getList()[2] = 3;

    return 0;
}
```

Overloading operator[]

However, a better solution in this case is to overload the subscript operator ([]) to allow access to the elements of m_list. The subscript operator is one of the operators that must be overloaded as a member function. An overloaded operator[] function will always take one parameter: the subscript that the user

places between the hard braces. In our IntList case, we expect the user to pass in an integer index, and we'll return an integer value back as a result.

```
class IntList
{
private:
    int m_list[10]{};

public:
    int& operator[] (int index);
};

int& IntList::operator[] (int index)
{
    return m_list[index];
}
```

Now, whenever we use the subscript operator ([]) on an object of our class, the compiler will return the corresponding element from the m_list member variable! This allows us to both get and set values of m_list directly:

```
    IntList list{};
    list[2] = 3; // set a value
    std::cout << list[2] << '\n'; // get a value

    return 0;
```

This is both easy syntactically and from a comprehension standpoint. When list[2] evaluates, the compiler first checks to see if there's an overloaded operator[] function. If so, it passes the value inside the hard braces (in this case, 2) as an argument to the function.

Note that although you can provide a default value for the function parameter, actually using operator[] without a subscript inside is not considered a valid syntax, so there's no point.

Why operator[] returns a reference

Let's take a closer look at how list[2] = 3 evaluates. Because the subscript operator has a higher precedence than the assignment operator, list[2] evaluates first. list[2] calls operator[], which we've defined to return a reference to list.m_list[2]. Because operator[] is returning a reference, it returns the actual list.m_list[2] array element. Our partially evaluated expression becomes list.m_list[2] = 3, which is a straightforward integer assignment.

In lesson 1.3 -- Introduction to variables, you learned that any value on the left hand side of an assignment statement must be an l-value (which is a variable that has an actual memory address). Because the result of operator[] can be used on the left hand side of an assignment (e.g. list[2] = 3), the return value of operator[] must be an l-value. As it turns out, references are always l-values, because you can only take a reference of variables that have memory addresses. So by returning a reference, the compiler is satisfied that we are returning an l-value.

Consider what would happen if operator[] returned an integer by value instead of by reference. list[2] would call operator[], which would return the *value of* list.m_list[2]. For example, if m_list[2] had the value of 6, operator[] would return the value 6. list[2] = 3 would partially evaluate to 6 = 3, which makes no sense! If you try to do this, the C++ compiler will complain:

C:VCProjectsTest.cpp(386) : error C2106: '=' : left operand must be l-value

Dealing with const objects

In the above IntList example, operator[] is non-const, and we can use it as an l-value to change the state of non-const objects. However, what if our IntList object was const? In this case, we wouldn't be able to call the non-const version of operator[] because that would allow us to potentially change the state of a const object.

The good news is that we can define a non-const and a const version of operator[] separately. The non-const version will be used with non-const objects, and the const version with const-objects.

```
#include <iostream>

class IntList
{
private:
    int m_list[10]{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }; // give this class some initial state for this example

public:
    int& operator[] (int index);
    const int& operator[] (int index) const;
};

int& IntList::operator[] (int index) // for non-const objects: can be used for assignment
{
    return m_list[index];
}

const int& IntList::operator[] (int index) const // for const objects: can only be used for access
{
    return m_list[index];
}

int main()
{
    IntList list{};
    list[2] = 3; // okay: calls non-const version of operator[]
    std::cout << list[2] << '\n';

    const IntList clist{};
    clist[2] = 3; // compile error: calls const version of operator[], which returns a const reference.  Cannot assign to this.
    std::cout << clist[2] << '\n';

```

```
33      return 0;
34  }
```

If we comment out the line clist[2] = 3, the above program compiles and executes as expected.