OBJECT ORIENTED PROGRAMMING LAB

Lab Manual # 09

Const Keyword

Operator Overloading

Instructor: Hurmat Hidayat

Semester Spring, 2022

Course Code: CL1004

Fast National University of Computer and Emerging Sciences Peshawar

Department of Computer Science

OBJECT ORIENTED PROGRAMMING LANGUAGE

Table of Contents

Const Keyword

It is the const keywords used to define the constant value that cannot change during program execution. It means once we declare a variable as the constant in a program, the variable's value will be fixed and never be changed. If we try to change the value of the const type variable, it shows an error message in the program.

Const variable

It is a const variable used to define the variable values that never be changed during the execution of a program. And if we try to modify the value, it throws an error.

```
const data_type variable_name;
```

Example:

```
#include <iostream>
#include <conio.h>
using namespace std;

int main ()
{
     // declare the value of the const
     const int num = 25;
     num = num + 10;
     return 0;
}

/*
Output:
It shows the compile-time error because we update the assigned value of
the num 25 by 10.
*/
```

const Data Member

Data members are like the variable that is declared inside a class, but once the data member is initialized, it never changes, not even in the constructor or destructor. The constant data member is initialized using the const keyword before the data type inside the class. The const data members cannot be assigned the values during its declaration; however, they can assign the constructor values.

Example:

```
#include <iostream>
using namespace std;
// create a class ABC
class ABC
{

public:
 // use const keyword to declare const data member
const int A;
// create class constructor
ABC ( int y) : A(y)
{
cout << " The value of y: " << y << endl;
}
};
int main ()
{
ABC obj( 10); // here 'obj' is the object of class ABC
cout << " The value of constant data member 'A' is: " << obj.A << endl;
// obj.A = 5; // it shows an error.
// cout << obj.A << endl;
return 0;
}

/*
Output:
The value of y: 10
The value of constant data member 'A' is: 10
*/
```

const Member Functions

- A const member function guarantees that it will never modify any of its class's member data.
- A function is made into a constant function by placing the keyword const after the declarator but before the function body.
- Member functions that do nothing but acquire data from an object are obvious candidates for being made const, because they don't need to modify any data.

Example:

```
class aClass {
private:
     int alpha;
public:
     void nonFunc()                //non-const member function
     { alpha = 99; }               //OK

     void conFunc() const          //const member function
     { alpha = 99; }               //ERROR: can't modify a member
};
```

The non-const function nonFunc() can modify member data alpha, but the constant function conFunc() can't. If it tries to, a compiler error results.

const Member Function Arguments

If an argument is passed to an ordinary function by reference, and you don't want the function to modify it, the argument should be made const in the function declaration (and definition). This is true of member functions as well. For example:

```
Distance Distance::add_dist(const Distance& d2) const
{
     Distance temp;                        //temporary variable
     // feet = 0;                          //ERROR: can't modify this
     // d2.feet = 0;                       //ERROR: can't modify d2
     temp.inches = inches + d2.inches;     //add the inches
     if(temp.inches >= 12.0)               //if total exceeds 12.0,
     {                                     //then decrease inches
            temp.inches -= 12.0;           //by 12.0 and
            temp.feet = 1;                 //increase feet
     }                                     //by 1
     temp.feet += feet + d2.feet;          //add the feet
     return temp;
}
```

Example : Distance

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

class Distance {

private:
     int feet;
     float inches;

public:

     //constructor (no args)
     Distance() : feet(0), inches(0.0) { }

     //constructor (two args)
     Distance(int ft, float in) : feet(ft), inches(in) {
     }

     //get length from user
     void getdist() {

            cout << "\nEnter feet: ";
            cin >> feet;

            cout << "Enter inches: ";
            cin >> inches;
     }

     //display distance
     void showdist() const {
            cout << feet << "\'-" << inches << '\"';
     }

     Distance add_dist(const Distance&) const; //const member function with
const function argument

};

Distance Distance::add_dist(const Distance &d2) const
{
     Distance temp;                  //temporary variable
     // feet = 0;                    //ERROR: can't modify this
```

```
    // d2.feet = 0;                //ERROR: can't modify d2

    temp.inches = inches + d2.inches;     //add the inches

    if (temp.inches >= 12.0)                    //if total exceeds 12.0,
    {
    //then decrease inches
           temp.inches -= 12.0;                 //by 12.0 and
           temp.feet = 1;                           //increase feet
    }
    //by 1

    temp.feet += feet + d2.feet;          //add the feet

    return temp;

}

int main() {
    Distance dist1, dist3;                      //define two lengths
    Distance dist2(11, 6.25);             //define, initialize dist2
    dist1.getdist();                            //get dist1 from user

    dist3 = dist1.add_dist(dist2);        //dist3 = dist1 + dist2

    //display all lengths
    cout << "\ndist1 = "; dist1.showdist();
    cout << "\ndist2 = "; dist2.showdist();
    cout << "\ndist3 = "; dist3.showdist();
    cout << endl;
    return 0;
}
/*
Output:
Enter feet: 44
Enter inches: 3

dist1 = 44'-3"
dist2 = 11'-6.25"
dist3 = 55'-9.25”
*/
```

const Objects

we can apply const to variables of basic types such as int to keep them from being modified. In a similar way, we can apply const to objects of classes. When an object is declared as const, you can't modify it. It follows that you can use only const member functions with it, because they're the only ones that guarantee not to modify it.

```
const Distance football(300, 0);
football.getdist();                    //ERROR: getdist() not const
```

Operator Overloading

In C++, we can change the way operators work for user-defined types like objects and structures. This is known as **operator overloading**. For example, Suppose we have created three objects c1, c2 and result from a class named Complex that represents complex numbers. Since operator overloading allows us to change how operators work, we can redefine how the + operator works and use it to add the complex numbers of c1 and c2 by writing the following code:

```
result = c1 + c2;
```

instead of something like

```
result = c1.addNumbers(c2);
```

This makes our code intuitive and easy to understand.
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 or structure whose 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.

Operator Overloading in Unary Operators

Unary operators operate on only one operand. The increment operator ++ and decrement operator -- are examples of unary operators.

Example:

```
// Overload ++ when used as prefix

#include <iostream>
using namespace std;

class Count {
   private:
    int value;

   public:
    // Constructor to initialize count to 5
    Count() : value(5) {}

    // Overload ++ when used as prefix
    void operator ++ () {
        ++value;
    }
    void display() {
        cout << "Count: " << value << endl;
    }
};
int main() {
    Count count1;
    // Call the "void operator ++ ()" function
    ++count1;
    count1.display();
    return 0;
}
```

Output:

```
Count: 6
```

Here, when we use ++count1;, the void operator ++ () is called. This increases the value attribute for the object count1 by 1.

Note: When we overload operators, we can use it to work in any way we like. For example, we could have used ++ to increase value by 100.

The above example works only when ++ is used as a prefix. To make ++ work as a postfix we use this syntax.

```
void operator ++ (int) {
    // code
}
```

Notice the int inside the parentheses. It's the syntax used for using unary operators as postfix; it's not a function parameter.

Example:

```
// Overload ++ when used as prefix and postfix

#include <iostream>
using namespace std;

class Count {
   private:
    int value;

   public:

    // Constructor to initialize count to 5
    Count() : value(5) {}


    // Overload ++ when used as prefix
    void operator ++ () {
        ++value;
    }


    // Overload ++ when used as postfix
    void operator ++ (int) {
        value++;
    }

    void display() {
        cout << "Count: " << value << endl;
    }
};
```

```
int main() {
    Count count1;

    // Call the "void operator ++ (int)" function
    count1++;
    count1.display();

    // Call the "void operator ++ ()" function
    ++count1;

    count1.display();
    return 0;
}
```

Output:

```
Count: 6
Count: 7
```

The **Example** works when ++ is used as both prefix and postfix. However, it doesn't work if we try to do something like this:

```
Count count1, result;

// Error
result = ++count1;
```

This is because the return type of our operator function is void. We can solve this problem by making Count as the return type of the operator function.

Return Value from Operator Function (++ Operator)

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

class Count {
   private:
    int value;

   public
       :
    // Constructor to initialize count to 5
    Count() : value(5) {}
```

```
    // Overload ++ when used as prefix
    Count operator ++ () {
        Count temp;

        // Here, value is the value attribute of the calling object
        temp.value = ++value;

        return temp;
    }

    // Overload ++ when used as postfix
    Count operator ++ (int) {
        Count temp;

        // Here, value is the value attribute of the calling object
        temp.value = value++;

        return temp;
    }

    void display() {
        cout << "Count: " << value << endl;
    }
};

int main() {
    Count count1, result;

    // Call the "Count operator ++ ()" function
    result = ++count1;
    result.display();

    // Call the "Count operator ++ (int)" function
    result = count1++;
    result.display();

    return 0;
}
```

Output

```
Count: 6
```

```
Count: 6
```

Operator Overloading in Binary Operators

Binary operators work on two operands. For example, result = num + 9;

Here, + is a binary operator that works on the operands num and 9. When we overload the binary operator for user-defined types by using the code:
obj3 = obj1 + obj2;

The operator function is called using the obj1 object and obj2 is passed as an argument to the function.

Eaxmaple:

```
// C++ program to overload the binary operator +
// This program adds two complex numbers
#include <iostream>
using namespace std;

class Complex {
   private:
    float real;
    float imag;

   public:
    // Constructor to initialize real and imag to 0
    Complex() : real(0), imag(0) {}

    void input() {
        cout << "Enter real and imaginary parts respectively: ";
        cin >> real;
        cin >> imag;
    }

    // Overload the + operator
    Complex operator + (const Complex& obj) {
        Complex temp;
        temp.real = real + obj.real;
        temp.imag = imag + obj.imag;
        return temp;
    }
    void output() {
        if (imag < 0)
            cout << "Output Complex number: " << real << imag << "i";
        else
```

```
            cout << "Output Complex number: " << real << "+" << imag <<
"i";
    }
};

int main() {
    Complex complex1, complex2, result;

    cout << "Enter first complex number:\n";
    complex1.input();

    cout << "Enter second complex number:\n";
    complex2.input();

   // complex1 calls the operator function
   // complex2 is passed as an argument to the function
    result = complex1 + complex2;
    result.output();

    return 0;
}
```

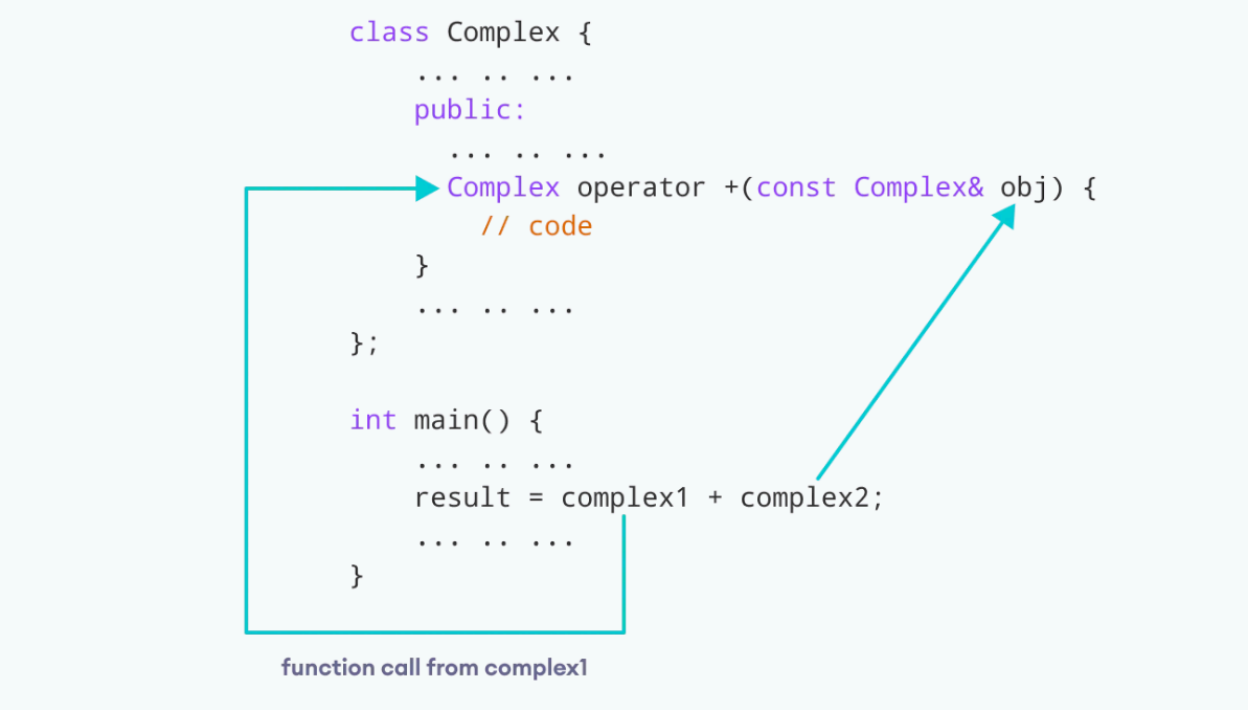
Output:

```
Enter first complex number:
Enter real and imaginary parts respectively: 9 5
Enter second complex number:
Enter real and imaginary parts respectively: 7 6
Output Complex number: 16+11i
```

In this program, the operator function is:

```
Complex operator + (const Complex& obj) {
    // code
}
```

- using & makes our code efficient by referencing the complex2 object instead of making a duplicate object inside the operator function.
- using const is considered a good practice because it prevents the operator function from modifying complex2.



Things to Remember in C++ Operator Overloading

1. Two operators = and & are already overloaded by default in C++. For example, to copy objects of the same class, we can directly use the = operator. We do not need to create an operator function.
2. Operator overloading cannot change the precedence and associativity of operators. However, if we want to change the order of evaluation, parentheses should be used.
3. There are 4 operators that cannot be overloaded in C++. They are:
    a. :: (scope resolution)
    b. . (member selection)
    c. .* (member selection through pointer to function)
    d. ?: (ternary operator)

References

https://www.programiz.com/cpp-programming/operator-overloading
https://www.javatpoint.com/const-keyword-in-cpp