



Table of Contents

- Chapter 1: **Overloading**
 - 1.1: Overloading – I
 - 1.1.1: Learning Objectives
 - 1.1.2: Overloading Member Functions
 - 1.1.2.1: Example – Member Function Overloading
 - 1.1.2.2: Matching Overloaded Function Calls
 - 1.1.3: Overloading Operators
 - 1.1.3.1: Friend Functions
 - 1.2: Overloading – II
 - 1.2.1: Learning Objectives
 - 1.2.2: Overloading Binary Operators
 - 1.2.2.1: Overloading Arithmetic Operators
 - 1.2.2.2: Overloading the Equality Operator
 - 1.2.3: Overloading I/O Operators
 - 1.2.3.1: Overloading >>
 - 1.2.3.2: Overloading <<
 - 1.2.4: Overloading Unary Operators
 - 1.2.5: Operator Overloading using Member Functions
 - 1.2.5.1: Overloading the Assignment Operator
 - 1.2.6: Overloading Increment and Decrement Operators
 - 1.2.6.1: Overloading Prefix Forms
 - 1.2.6.2: Overloading Postfix Forms
- Summary
- Exercises
- Glossary
- References

1.1 Overloading

Overloading refers to the use of the same thing for different purposes. It is a programming language feature that allows you to declare multiple functions with the same name as long as they have a different signature. Overloading is one of the most powerful features of C++.

Overloading member functions of a class lets you create a family of functions that share the same name but perform different tasks. Overloading an operator lets you define how to perform a certain operation when the operator is used on one or more objects.

Overloading can lower a program's complexity significantly while introducing very little additional risk. These concepts will help you write simpler programs that have enhanced readability and maintainability.

1.1.1 Learning Objectives

After reading this chapter you should be able to:

- Employ overloaded functions and operators.
 - Implement overloaded class member functions to develop a family of functions.
 - Describe how overloaded function calls are resolved.
 - Describe what an ambiguous match for an overloaded function is and why it can occur.
 - State the purpose and use of overloaded operators.
 - Create friend functions to allow non member functions access to private members of a class.
 - Enumerate the advantages of function and operator overloading.

1.1.2 Overloading Member Functions

Recall that a function is said to be overloaded when the same name is given to a different function. However, the two functions should have different signatures. They must differ in at least one of the following:

- The number of parameters
- The data type of parameters
- The order of appearance of parameters

Member functions of a class can be overloaded just like normal functions.

1.1.2.1 Example – Member Function Overloading

Consider the following member function of a class named Overload that adds two integers

```
int Overload::add(int a, int b)
{
    return a + b;
}
```

Overloading

This function can be invoked to sum two integers but if you need to add two floating point numbers, you cannot use this function.

If this function is invoked as `cout << e.add(13.4, 15.32)` what would happen?

The compiler will successfully compile the program (though with warnings!!). The output will be 28 and not 28.72 as expected. This is because when the function "add" is called with floating point arguments, the arguments are converted to integer parameters causing the numbers to lose their fractional part.

One way to solve this issue is to declare another function to add two floating point numbers:

```
double Overload::add2(double a, double b)
{ return a+b; }
```

Then when we wish to add integers, we invoke *add* and when we wish to add real numbers, we invoke *add2*. The obvious disadvantage is that we need to remember two function names – one to add integers and another to add real numbers.

Function overloading provides a better solution. Using function overloading, we can declare another *add()* function that takes double parameters:

```
double Overload::add(double a, double b)
{ return a+b; }
```

We can have as many overloaded versions of *add* as we need as long as each of the *add()* function has unique parameters:

```
int Overload::add(int a, int b) // to add two integers
double Overload::add(double a, double b) //to add two floating point numbers
double Overload::add(int a, double b) //to add an integer and a floating point number
```

When the function call to *add()* is made, the compiler compares the data types of the arguments and those of the function parameters to decide which version to call. When the function call is *add(13,15)* the integer version is invoked and when the function call is *add(13.4, 15.32)*, the floating point version is invoked.

It is also possible to overload the *add()* function with a differing number of parameters or with a different order of appearance of the parameters:

```
double Overload::add(double a, double b, double c)
{ return a+b+c; } //to add three floating point numbers

double Overload::add(double a, int b); //to add a floating point number and an integer
```

Value addition: Common Coding Errors

Heading text – Function Overloading - BEWARE!!

Note that the function's return type is not considered when overloading functions. For example, the following function declarations would generate a compiler error:

Overloading

```
int    add(int a, int b)
double add(int a, int b)           // error – ambiguous declaration
```

Since these two functions have the same number and type of parameter, they differ only in the return type; the second declaration will be treated as an erroneous redeclaration of the first.

Source: Self Made

Value addition: Source Code

Heading text – FunctionOverloading

```
/* This program demonstrates function overloading */

#include <iostream>
using namespace std;

class Overload {
public:
    int add(int,int);
    double add(double,double);
    double add(double,double,double);
    double add(int,double);
    double add(double,int);
};

int Overload::add( int a, int b)
{
    cout << "Adding two integers ("<<a<<","<<b<<"): ";
    return a+b;
}

double Overload::add(double a, double b)
{ cout << "Adding two floating point numbers("<<a<<","<<b<<"): ";
  return a+b;
}

double Overload::add(double a, double b, double c)
{ cout << "Adding three floating point numbers ("<<a<<","<<b<<","<<c<<"): ";
  return a+b+c;
}

double Overload::add(int a, double b)
{ cout<<"Adding an integer and a floating point number ("<<a<<","<<b<<"): ";
  return a+b;
}

double Overload::add(double a, int b)
{ cout << "Adding a floating point number and an integer ("<<a<<","<<b<<"): ";
  return a+b;
}
```

```

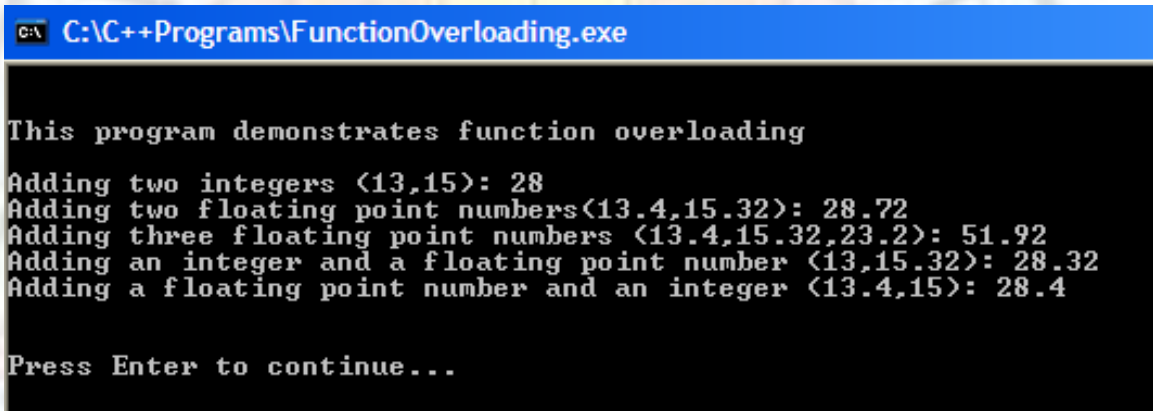
}

int main()
{
    Overload O;

    cout << "\n\nThis program demonstrates function overloading\n\n";
    cout << O.add(13,15)<<endl;
    cout << O.add(13.4, 15.32)<<endl;
    cout << O.add(13.4, 15.32, 23.2)<<endl;
    cout << O.add(13, 15.32)<<endl;
    cout << O.add(13.4, 15)<<endl;

    cout << "\n\nPress Enter to continue...";
    getchar();
    return 0;
}

```



```

C:\C++Programs\FunctionOverloading.exe

This program demonstrates function overloading
Adding two integers <13,15>: 28
Adding two floating point numbers<13.4,15.32>: 28.72
Adding three floating point numbers <13.4,15.32,23.2>: 51.92
Adding an integer and a floating point number <13,15.32>: 28.32
Adding a floating point number and an integer <13.4,15>: 28.4

Press Enter to continue...

```

Source: Self Made

1.1.2.2 Matching Overloaded Function Calls

When a call is made to an overloaded function, the actual arguments are matched with the parameter lists of the functions one by one. One of the following three possibilities can occur:

- A match is found. The matched function is invoked.
- No match is found. The compiler tries to find a match in the following order:

Overloading

1. The compiler tries to find a match by automatic type promotion (For example, char is promoted to int, float is promoted to double and so on).
 2. If no match is found even after promotion, the compiler tries to find a match through standard conversion (For example, int to float, enum to float and so on).
 3. If no match is found even after standard conversion, the compiler tries to find a match through user defined conversions (For example, if user has declared a class called Y and has provided a conversion of class Y to int).
 4. If a match is still not found, a compile time error is generated.
- An ambiguous match is found. The arguments match more than one overloaded function. A compile time error is generated.

If there are multiple arguments, C++ applies the matching rules to each argument in turn. The function chosen is the one for which each argument matches at least as well as all the other functions, with at least one argument matching better than all the other functions.

The overload resolution is static, meaning that the compiler resolves overloaded function calls at compile-time. This static overloading resolution has two advantages: The compiler is able to detect incorrect arguments, and there's no performance overhead resulting from runtime resolution.

Value addition: FAQ – Ambiguous Matches

Heading text – How can an overloaded function match multiple candidates?

If a function call matches multiple candidates because of standard conversion or user-defined conversion, an ambiguous match will result.

Consider for example, the following overloaded function declarations:

```
void display(unsigned int n);
```

```
void display(float n);
```

When the function is invoked with `display(0)`, the compiler tries to find a matching function. Since there is no `display(int)` function, the compiler tries to find a match by automatic type promotion or standard conversion. 0 can be converted either to unsigned int or to float. Since all standard conversions are equal, the function call results in an ambiguous match as it can be resolved either to the first function or to the second. This results in a compile time error.

Source: Self Made

Value addition: Common Coding Errors

Heading text – Function Overloading

Declaring a typedef does not introduce a new type — consequently, the following the two declarations of DisplayString() are considered identical:

```
typedef char *mystring;
void DisplayString(mystring abc);
void DisplayString(char *abc);
```

Source: Self Made

1.1.3 Overloading Operators

Operator overloading is similar to function overloading. In fact, in C++ operators are implemented as functions. You can define new functionality for existing operators by overloading them. In C++ this can be done by using the keyword **operator**.

For example, you can overload the + operator to concatenate your user-defined string class objects or to add two Complex class objects together.

To add integers, we use the binary operator + as below:

```
int a, b, c;
c = a + b;
```

By overloading the + operator to add functionality for complex numbers, you can add complex number objects in the same manner:

```
Complex a, b, c;
c = a + b;
```

The general syntax for defining an operator overloading is as follows:

```
return_type classname :: operator operator symbol(argument)
{
    .....
    statements;
}
```

In the above:

- return_type – is the data type returned by the function
- class name - is the name of the class
- operator – is the keyword
- operator symbol – is the symbol of the operator which is being overloaded
- :: - is the scope resolution operator

When evaluating an expression with operators, C++ looks at the operands around the operator to see what type they are. If *all* operands are built-in types, C++ calls a built-in routine. If *any* of the operands are user data types (For example, one of your classes), it looks to see whether the class has an overloaded operator function that it can call. If the compiler finds an overloaded operator whose parameters match the types of the operands, it calls that function. Otherwise, it produces a compiler error.

Operators can be overloaded using friend functions or by using member functions. Using which way is a programmer's preference.

However, if the operator does not change the operands a friend function is preferred to overload the operator (for example +). If the operator changes the operands, a member function would be more suitable (for example ++).

1.1.3.1 Friend Functions

You have learnt in earlier lessons on access specifiers that when data members are declared as private inside a class, they are inaccessible outside the class. The private members are available for use only to member functions. However, there might be situations, such as in operator overloading, where you would like to allow a non member function or an external class access to private and protected data members of a class. For handling such cases, you can use the concept of **friend functions**.

A class can allow non-member functions and other classes to access its own private and protected data, by making them friends. A friend function is just like a normal function but it can access the private members of a class as though it were a member of that class. A friend function may or may not be a member of another class. To declare a friend function, simply use the *friend* keyword in front of the prototype of the function you wish to be a friend of the class. It does not matter whether you declare the friend function in the private or public section of the class.

```
class Example{
private:
    int data;
public:
    Example() { data = 0; }
    void add(int d) { data = data + d; }
    void print() { cout << " data = " << data << endl;}
    friend void reset(Example &e); //make the reset function a friend function
};

void reset(Example &e) {
    e.data = 0; //non-member function can access private member
               // of Example class as it is a friend function of that class
}
```

When the operator does not modify its operands, a friend function is the best way to overload the operator, otherwise we use member function to overload the operator.

Value addition: FAQ

Heading text – Why should I overload functions or operators?

Overloading functions and operators has many advantages including the following:

- **Enhanced Readability:** The program code is compact and easier to read and understand.
- **Better Version Management:** When ever some new features are added in the new version, function over loading can be used to add a new function with

support for new version without disturbing the existing functions with features of previous version.

- **Extensibility:** An operator will act differently depending on the operands provided. Operator is not limited to work only with primitive Data Type

Source: Self Made

1.2 Overloading - II

In the previous lesson you learnt that you can create multiple class member functions with the same name but with different data types, number or order of parameters. In this lesson, you will learn operator overloading that allows you to define how operators (such as +, -, ==, ++, and so on) should operate on user defined data types.

1.2.1 Learning Objectives

After reading this chapter you should be able to:

- Implement polymorphic behavior through operator overloading.
 - Implement operator overloading through member functions
 - Implement operator overloading through friend functions
 - Overload the arithmetic, I/O, assignment and equality operators.
 - Overload the unary increment and decrement operators in prefix and postfix forms.
 - Identify which C++ operators can be overloaded.

1.2.2 Overloading Unary Operators

The unary operators (+, - and !) operate on only one operand. Because none of these operators change their operands, we can implement them as friend functions with just one parameter. To overload such an operator, simply declare the function, give it one parameter of the type of the operands, select an appropriate return type, then make the function a friend function of the class.

Let's take a look at how to implement the unary minus operator (-) on the Complex class:

```
Complex operator- (Complex C) {
    Complex temp;
    temp.real = -C.real;
    temp.imag = -C.imag;
    return temp;
}
```

The overloaded not (!) operator- takes one parameter of type Complex, and returns a value of type Complex. The function will return true when its argument has a 0 in both its real and imaginary parts – false otherwise.

```
bool operator! (Complex C) {
    return (C.real == 0 && C.imag == 0);
}
```

1.2.3 Overloading Binary Operators

Most binary operators do not change their operands so we can use friend functions to overload them. To overload such an operator, simply declare the function, give it two parameters of the type of the operands, select an appropriate return type, then make the function a friend function of the class.

1.2.3.1 Overloading Arithmetic Operators

In this example, we will overload the + operator to add complex numbers by declaring a friend function in the Complex class.

```
class Complex
{ private:
    double real;
    double imag;
public:
    Complex() {real = imag = 0.0; }
    ....
    friend Complex operator+(Complex C1, Complex C2) ;
};
```

Then we write the non member friend function operator+.

```
Complex operator+ (Complex C1, Complex C2) {
    Complex temp;
    temp.real = C1.real + C2.real;
    temp.imag = C1.imag + C2.imag;
    return temp;
}
```

To add two Complex objects together, we need to add their real and imaginary parts separately. Because the overloaded operator+() function is a friend of the Complex class, we can access the private real and imag members. Also, because both real and imag are double, C++ knows how to add integers together using the built-in version of the plus operator that works with double operands, we can simply use the + operator to do the adding.

The function call will be invoked when such a statement is executed:

```
C3 = C1 + C2;           //C1,C2,C3 are objects of the type Complex
```

We can overload the other arithmetic operators (-, *, /) operators similarly.

1.2.3.2 Overloading the Equality Operator

The == operator can be overloaded just like the arithmetic operators, it is also a binary operator. The == operator should return true when both its argument have the same real and imaginary parts – false otherwise.

```
bool operator==(Complex C1, Complex C2) {
    return (C1.real == C2.real && C1.imag == C2.imag);
}
```

1.2.4 Overloading I/O Operators

Overloading the insertion operator<< for output and the extraction operator >> for input is similar to overloading operator+ (they are all binary operators), except that the parameter types are different. Overloading the operator<< and operator>> makes it easy to output your class to screen and to accept user input.

1.2.4.1 Overloading <<

For classes that have multiple member variables, printing each of the individual variables on the screen can be quite tedious. For example, to print the Complex numbers, you need to write a statement such as:

```
cout << C.real;
if (C.imag >= 0) cout << "+"; //Don't print a + sign if the imaginary part is -ve
cout <<C.imag<<"i";
```

It would be much easier if we could simply type:

```
cout << C1; //C1 is an object of the Complex class.
```

To make this possible we need to overload the binary << operator. For binary operators, we need two operands. What are the two operands for the << operator? The left operand is the cout object, and the right operand is the Complex class object – C1. cout is actually an object of type ostream. Therefore, our overloaded function will look like this:

```
ostream& operator<<(ostream &out, Complex &C);
```

Implementation of the operator<< is fairly straightforward -we can simply use operator<< to output the data member variables of the class Complex.

```
ostream& operator<<(ostream &out, Complex &C) {
    out << C.real;
    if (C.imag >= 0) cout << "+";
    cout <<C.imag<<"i";
    return out;
}
```

Why do we need to return an object of type ostream? The answer is that we do this so we can "chain" output commands together, such as

```
cout << C1 << endl<< C2 << endl;
```

1.2.4.2 Overloading >>

It is also possible to overload the input operator `>>` similar to overloading the output operator `<<`. The only difference is that instead of an output stream object, this time we use an object of type `istream`.

```
istream& operator>>(istream &in, Complex &C) {  
    in >> C.real;  
    in >> C.imag;  
    return in;  
}
```

To input a complex number, now we just need to write the more intuitive:
`cin >> C1;`

1.2.5 Operator Overloading using Member Functions

In the previous sections, we learnt that when the operator does not modify its operands, it's best to implement the overloaded operator as a friend function of the class. Operator overloading for operators that modify their operands is implemented using a member function of the class.

When overloading an operator using a member function:

- The leftmost operand of the overloaded operator must be an object of the class type.
- The leftmost operand becomes the implicit ***this** parameter. All other operands become function parameters.

Previously we have overloaded the `+` operator for complex numbers using a friend function. Now, let's take a look at how to overload the same operator using a member function.

```
Complex Complex::operator+ (Complex C) {  
    Complex temp;  
    temp.real = real + C.real;  
    temp.imag = imag + C.imag;  
    return temp;  
}
```

When we use a friend function for overloading the binary `+` operator, we have two parameters. However, when using a member function, we need only one parameter because the leftmost parameter becomes the implicit `*this` parameter in the member function version.

1.2.5.1 Overloading the Assignment Operator

Let us now overload the binary assignment `=` operator. We will use a member function since this operator changes the operand.

The assignment operator is used to copy the values from one object to another already existing object.

```
Complex Complex::operator= (Complex C) {  
    real = C.real;  
    imag = C.imag;  
    return *this;  
}
```

```
}
```

We need to return an object of type Complex so that we can "chain" assignment statements together, such as

```
C1 = C2 = C3;
```

If a unary operator is overloaded using a member function, there will be no parameters in the function declaration.

1.2.6 Overloading Increment and Decrement Operators

Because the increment and decrement operators modify their operands, they are best overloaded as member functions. However there is a difference in how we overload the prefix and postfix versions.

1.2.6.1 Overloading Prefix Forms

Prefix increment and decrement is overloaded exactly the same as normal unary operators.

```
Complex Complex::operator++ () {
    return Complex(++real,++imag);
}
```

The invoking object is incremented and the incremented result is returned. If the operator is invoked as `C2 = ++C1;` First C1 is incremented by the function; the result is returned which is then assigned to C2 – as expected.

Since we return an object of type Complex we can even "chain" multiple statements together. The prefix decrement can be implemented similarly.

1.2.6.2 Overloading Postfix Forms

The prefix and postfix versions of the increment operators have the same name and both take one parameter of the same type. To differentiate the two when overloading, C++ uses a "dummy variable" for the postfix operators. This argument is a fake integer parameter that only serves to distinguish the postfix version of increment/decrement from the prefix version.

```
Complex Complex::operator++ (int ) {
    return Complex(real++, imag++);
}
```

The invoking object is incremented after its original values are returned. If the operator is invoked as

```
C2 = C1++;
```

First C1 is assigned to C2 and then it is incremented– as expected.

Since we return an object of type Complex we can even "chain" multiple statements together. The postfix decrement can be implemented similarly.

Value addition: Source Code
Heading text OperatorOverloading

Overloading

```
/* This program demonstrates operator overloading
for complex number objects*/

#include <iostream>
using namespace std;

class Complex
{ private:
    double real;
    double imag;
public:
    Complex() {real = imag = 0.0; }
    Complex(double r, double i) {real =r; imag = i; }
    Complex operator=(Complex C);
    Complex operator++();
    Complex operator--();
    Complex operator++(int);
    Complex operator--(int);

    friend Complex operator+(Complex C1, Complex C2) ;
    friend Complex operator-(Complex C1, Complex C2) ;
    friend Complex operator*(Complex C1, Complex C2) ;
    friend Complex operator/(Complex C1, Complex C2) ;
    friend ostream& operator<<(ostream &out, Complex &C);
    friend istream& operator>>(istream &in, Complex &C);
    friend Complex operator-(Complex C) ;
    friend bool operator!(Complex C) ;
    friend bool operator==(Complex C1, Complex C2);
};

Complex operator+ (Complex C1, Complex C2) {
    Complex temp;
    temp.real = C1.real + C2.real;
    temp.imag = C1.imag + C2.imag;
    return temp;
}

Complex operator- (Complex C1, Complex C2) {
    Complex temp;
    temp.real = C1.real - C2.real;
    temp.imag = C1.imag - C2.imag;
    return temp;
}

Complex operator* (Complex C1, Complex C2) {
    Complex temp;
    temp.real = C1.real*C2.real - C1.imag*C2.imag;
    temp.imag = C1.real*C2.imag + C1.imag*C2.real;
    return temp;
}

Complex operator/ (Complex C1, Complex C2) {
    Complex temp1,temp2,conjugate;
```

Overloading

```
    conjugate.real = C2.real;
    conjugate.imag = -C2.imag;
    temp1 = C1 * conjugate;
    temp2 = C2 * conjugate;
    temp1.real = temp1.real/temp2.real;
    temp1.imag = temp1.imag/temp2.real;
    return temp1;
}

Complex Complex::operator= (Complex C) {
    real = C.real;
    imag = C.imag;
    return *this;
}

Complex Complex::operator++ () {
    return Complex(++real, ++imag);
}

Complex Complex::operator-- () {
    return Complex(--real, --imag);
}

Complex Complex::operator++ (int ) {
    return Complex(real++,imag++);
}

Complex Complex::operator-- (int) {
    return Complex(real--, imag--);
}

ostream& operator<<(ostream &out, Complex &C) {
    out << C.real;
    if (C.imag >= 0)
        cout << "+";
    cout <<C.imag<<"i";
    return out;
}

istream& operator>>(istream &in, Complex &C) {
    in >> C.real;
    in >> C.imag;
    return in;
}

Complex operator- (Complex C) {
    Complex temp;
```


Overloading

```
        temp.real = -C.real;
        temp.imag = -C.imag;
        return temp;
    }

    bool operator! (Complex C) {
        return (C.real == 0 && C.imag == 0);
    }

    bool operator== (Complex C1, Complex C2) {
        return (C1.real == C2.real && C1.imag == C2.imag);
    }

int main()
{
    Complex C1, C2, C3, C4, C5;

    cout << "\nEnter a complex number    : ";
    cin >> C1;
    cout << "\nEnter another complex number: ";
    cin >> C2;
    cout << "\nC1 = " << C1 << endl << "C2 = " << C2 << endl;
    cout << "C3 = " << C3 << endl;
    if (!C3) cout << "\nC3 is equal to zero\n";
    else cout << "\nC3 is zero\n";
    if (C1 == C2) cout << "C1 is equal to C2\n\n";
    else cout << "C1 is not equal to C2\n\n";
    C3 = C1 + C2;
    cout << "After C3 = C1 + C2, C3 = " << C3 << endl;
    C4 = C1 - C2;
    cout << "After C4 = C1 - C2, C4 = " << C4 << endl;
    C3 = C1 * C2;
    cout << "After C3 = C1 * C2, C3 = " << C3 << endl;
    C4 = C1 / C2;
    cout << "After C4 = C1 / C2, C4 = " << C4 << endl;
    C5 = C1;
    cout << "After C5 = C1,    C5 = " << C5 << " C1 = " << C1 << endl;
    C5 = -C1;
    cout << "After C5 = -C1,    C5 = " << C5 << " C1 = " << C1 << endl;
    C5 = ++C2;
    cout << "After C5 = ++C2,    C5 = " << C5 << " C2 = " << C2 << endl;
    C5 = C2++;
    cout << "After C5 = C2++,    C5 = " << C5 << " C2 = " << C2 << endl;

    cout << "\n\nPress Enter to continue...";
    getchar();getchar();
    return 0;
}
```

```

C:\ C:\C++Programs\OperatorOverloading.exe

Enter a complex number      : 2 6
Enter another complex number: 4 1

C1 = 2+6i
C2 = 4+1i
C3 = 0+0i

C3 is equal to zero
C1 is not equal to C2

After C3 = C1 + C2, C3 = 6+7i
After C4 = C1 - C2, C4 = -2+5i
After C3 = C1 * C2, C3 = 2+26i
After C4 = C1 / C2, C4 = 0.823529+1.29412i
After C5 = C1, C5 = 2+6i C1 = 2+6i
After C5 = -C1, C5 = -2-6i C1 = 2+6i
After C5 = ++C2, C5 = 5+2i C2 = 5+2i
After C5 = C2++, C5 = 5+2i C2 = 6+3i

Press Enter to continue..._

```

Source: Self Made

Value addition: Common Coding Errors

Heading text Operator Overloading

1. Not all operators in C++ can be overloaded. The exceptions are: arithmetic conditional operator (? :), sizeof, scope (::), member selector (.), and member pointer selector (*).
2. At least one of the operands in an overloaded operator has to be a user defined type. For example, you cannot overloaded the + operator to add a double and an integer. You can use it however to add a user defined class object and an integer.
3. You can only overload operators that exist – you cannot create new operators. For example, you cannot overload the ** symbol to perform exponentiation.
4. You cannot change the arity (number of operands), precedence or associativity of operators.
5. Operators cannot be overloaded implicitly, they have to be overloaded explicitly. For example, overloading the == operator does not overload the += operator.

Source: Self Made

Summary

- Overloading is the feature provided by a programming language that allows declaration of multiple functions with the same name as long as they have a different signature.
- Overloaded functions must differ in at least one of the following - the number of parameters, the data type of parameters, or the order of appearance of parameters
- Member functions of a class can be overloaded just like normal functions.
- A function's return type is NOT considered when overloading functions. If two functions have the same number and type of parameter, the second declaration will be treated as an erroneous redeclaration of the first.
- When a call is made to an overloaded function, the actual arguments are matched with the parameter lists of the functions one by one.
- If no match to a function call is found, the compiler tries to find a match first by automatic type promotion, then through standard conversion and finally through user defined conversions. If a match is still not found, a compile time error is generated.
- In the case of an ambiguous match - the arguments match more than one overloaded function - a compile time error is generated.
- If there are multiple arguments, C++ applies the matching rules to each argument in turn.
- Operator overloading allows you to define new functionality for existing operators by using the keyword **operator**.
- The general syntax for defining an operator overloading is as follows:

```
return_type classname::operator operator symbol(argument list)
{
    .....
    statements;
}
```

- Overloading functions and operators has many advantages including enhanced readability, better version management and extensibility.
- Operators can be overloaded using friend functions or by using member functions.
- If the operator does not change the operands a friend function is preferred to overload the operator. If the operator changes the operands, a member function would be more suitable.
- A friend function is just like a normal function but it can access the private members of a class as though it were a member of that class.
- To declare a friend function, simply use the *friend* keyword in front of the prototype of the function you wish to be a friend of the class.
- To overload an operator using friend function, declare the function, give it the parameters of the type of the operands, select an appropriate return type, then make the function a friend function of the class.
- When overloading an operator using a member function, the leftmost operand of the overloaded operator must be an object of the class type. The leftmost operand becomes the implicit ***this** parameter. All other operands become function parameters.
- Overloading the operator<< and operator>> makes it easy to output your class to screen and to accept user input.
- Prefix increment and decrement is overloaded exactly the same as normal unary operators.
- To differentiate the prefix and postfix forms or the increment/decrement operators when overloading, C++ uses a "dummy variable" for the postfix operators.
- Not all operators in C++ can be overloaded. The exceptions are: arithmetic conditional operator (? :), sizeof, scope (::), member selector (.), and member pointer selector (.*)

- At least one of the operands in an overloaded operator has to be a user defined type.
- You can only overload operators that exist – you cannot create new operators.
- You cannot change the precedence, arity or associativity of operators.

Exercises

- 1.1 What are the advantages of function overloading and operator overloading?
- 1.2 The C language does not support function overloading. Consequently there are many functions to find the absolute value of a number. The function **abs()** returns the absolute value of an integer, **labs()** returns the absolute value of a long integer, and **fabs()** returns the absolute value of a floating-point value. Write an overloaded function family **abs()** in C++ so that it can find the absolute value for any numeric type with the function **abs()**.
- 1.3 Write an overloaded version of the **min()** function that returns either the smaller of two integers, two characters or two floating point numbers.
- 1.4 Overload the **swap()** function so that you can use it to exchange either integers, doubles or characters.
- 1.5 Identify the error when the following function prototypes are declared.
 int square(double x);
 double square(double x);
Why isn't function overloading in this way permitted?
- 2.1 How do you differentiate between overloading the prefix and postfix increment operators?
- 2.2 Determine the dummy int constant value that your compiler passes for postfix operator++ and operator--.
- 2.3 Is it legal in C++ to overload the operator++ so that it decrements a value in your class?
- 2.4 Write a SimpleBox class declaration with one private data member: side.
 - Include a default constructor, a destructor, and accessor methods for side.
 - Write the implementation of the default constructor, initializing side with the value 5.
 - Overload the constructor so that it takes a value as its parameter and assigns that value to side.
 - Create a prefix and postfix increment operator for your SimpleBox class that increments its side.
 - Overload the << and the >> I/O operators for the SimpleBox class.
 - Provide an assignment operator for SimpleBox.
 - Provide an implementation of the binary operator + that takes two SimpleBox objects and creates a larger SimpleBox whose side is the sum of the sides of the two operands.
 - Write an implementation for the ! operator that returns true if the side of its operand is 0, false otherwise.
 - Write a main that creates two SimpleBox objects. Instantiate one with the value 9. Take user input for the other with the >> operator. Use the ! operator to determine if the second SimpleBox has side 0. If yes, then prompt the user again for its side. Print the values of the sides of the two SimpleBox objects using the << operator. Call the increment operator on each and again print their values. Create a larger SimpleBox object by adding the two previous boxes using the + operator and print its value. Finally, assign the second object to the first and print their values.

Glossary

Ambiguous Match - The arguments in a function call match more than one overloaded function.

Friend Function - A non member function that can access the private members of a class as though it were a member of that class.

Operator Overloading - A feature that allows defining new functionality for existing operators by overloading them. In C++ this can be done by using the keyword **operator**.

Overloading - A feature provided by a programming language that allows declaration of multiple functions with the same name as long as they have a different signature.

References

1. Works Cited

2. Suggested Reading

1. B. A. Forouzan and R. F. Gilberg, Computer Science, A structured Approach using C++, Cengage Learning, 2004.
2. R.G. Dromey, How to solve it by Computer, Pearson Education
3. E. Balaguruswamy, Object Oriented Programming with C++ , 4th ed., Tata McGraw Hill
4. G.J. Bronson, A First Book of C++ From Here to There, 3rd ed., Cengage Learning.
5. Graham Seed, An Introduction to Object-Oriented Programming in C++, Springer
6. J. R. Hubbard, Programming with C++ (2nd ed.), Schaum's Outlines, Tata McGraw Hill
7. D S Malik, C++ Programming Language, First Indian Reprint 2009, Cengage Learning
8. R. Albert and T. Breedlove, C++: An Active Learning Approach, Jones and Bartlett India Ltd.

3. Web Links

- 1.1 <http://www.devarticles.com/c/a/Cplusplus/Operator-Overloading-in-C-plus/>
- 1.2 <http://www.codeproject.com/KB/cpp/cfraction.aspx>
- 1.3 [http://msdn.microsoft.com/en-us/library/5tk49fh2\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/5tk49fh2(VS.80).aspx)
- 1.4 http://www.deitel.com/articles/cplusplus_tutorials/20060204/
- 1.5 http://www.devhood.com/tutorials/tutorial_details.aspx?tutorial_id=502