**Chapter 8**

Table of Contents

[Operator Overloading 2](#_Toc49508948)

[Prefix Notation 3](#_Toc49508949)

[Nameless Temporary Objects 5](#_Toc49508950)

[Postfix Notation 6](#_Toc49508951)

[Overloading Binary Operators 7](#_Toc49508952)

[Concatenating Strings 9](#_Toc49508953)

[Safe Array Example 10](#_Toc49508954)

[Data Conversion 13](#_Toc49508955)

## Operator Overloading

In the class Counter that we created in a previous lecture, we had two functions like these:

c3.addCount(c1, c2);  
c3 = c1.addCount(c2);

C++

Both of these functions added the counter integer from two objects of the Counter class. It is pretty nice that we can do this, but it would be even more nice if we could simply write

c3 = c1 + c2;

C++

Using operator overloading, we can actually do this.

## Prefix Notation

Unary operators include ++ and --. We had a function in the Counter class that incremented the counter integer of an object. It was called like this:

c1.incCounter();

C++

What we want to do instead is this:

++c1;

C++

We are looking at the prefix incrementation method first. Inside the Counter class, we can define another function to overload the ++ operator.

void operator ++()  
{  
 count++;  
}

C++

Remember that we saw earlier that the only way for the compiler to distinguish between different overloaded functions with the same name is by looking at the data types and the number of arguments. The compiler will distinguish between different overloaded operators in the same way. If we increment an integer, the compiler uses the built-in routine to increment the integer, but if we increment an object of the counter class, it will use this function. If we had another class that also overloaded the same operator, the compiler would still be able to distinguish between the two separate functions based on the object which calls it.

This function can take parameters, but in this case, there are none. When we call the ++ operator with any object, like we did above, the data for that particular object is accessed and the counter integer for that particular object is incremented. Since the member function of an object can access the variables defined in the class for that object, this function requires no parameters.

The function also has a return type of void. This is a small defect since if we tried to write a statement like this:

c2 = ++c1;

C++

it would cause an error. This statement requires something to be returned from the function that is being called. To fix this, we can edit the previous function to make it like this:

Counter operator ++ ()  
{  
 counter++;  
 Counter temp;  
 temp.counter = counter;  
 return temp;  
}

C++

This function creates a new temporary object of the Counter class, and returns that object. Thus, we can now write both of these statements:

++c1;  
c2 = ++c1;

C++

## Nameless Temporary Objects

We can write the last function we created like this as well:

Counter operator ++ ()  
{  
 counter++;  
 return Counter (counter);  
}

C++

Here, in the return statement, we created an object of the Counter class, which set the value of the counter integer for that object using the previously defined constructor function, and then returned that temporary object. This object is called a nameless temporary object. It does not have a name and requires a constructor function to be defined inside that class.

## Postfix Notation

To use the statement

c1++;

C++

we need to defined another overloaded operator like this:

Counter operator ++ (int)  
{  
 return Counter (count++);  
}

C++

The ‘argument’ we have used in this function is not really an argument, and it does not actually represent an integer. This is simply the way the people who created the C++ language decided to differentiate between prefix and postfix operators.

## Overloading Binary Operators

For the statement

c3 = c1 + c2;

C++

we can use this function:

Counter operator + (Counter obj)  
{  
 return Counter (counter + obj.counter);  
}

C++

Here, the argument on the left of the operator is the object for which the operator is a member. Thus, we go to the operator overloading function for c1, take c2 as an argument, use the sum of their counter values as the constructor argument for the nameless temporary object, and return that object from the function, causing it to be set to c3. We can access the counter variable of c1 directly (without having to write c1.counter), since that is the object through which we called the function.

We could even have written a statement like this:

c4 = c1+c2+c3;

C++

Here, the overloaded + operator would first work with c1 and c2, get a resulting object, then work with that object and c3, and set the result to c4.

We can overload other operator types as well like logical operators (<, >, !=, ==) and assignment operators (=, +=, -=). Note however, that assignment operator functions do not require a return type.

C2 = c3;

C++

for the function

void operator = (Counter obj)  
{  
 counter = obj.counter;  
}

C++

This may seem a little confusing, but the object on the left of the assignment operator is actually calling the overloaded operator function using the object on the right of the operator as an argument. We are not returning something and then assigning it to something else, we are working with the assignment itself.

## Concatenating Strings

We cannot normally add character arrays (C-string variables) like this:

str3 = str2 + str1;

C++

However, if we use our own MyString class, we can then overload the + operator to do exactly this. The existing string class in C++ does this, albeit in a more sophisticated manner.

MyString operator + (MyString s const)  
{  
 MyString temp;  
 strcat(temp.array, array);  
 strcat(temp.array, s.array);  
 return temp;  
}

Notice a few things here. First, we are using the + operator to do something that is not strictly addition. This goes to show that we can actually make the operators do anything we want. There is nothing stopping us from overloading the increment operator we used earlier to decrement count, except perhaps a few choice words from anyone attempting to use our code. Next, notice that we have to use a MyString object as the argument. If we want to use some string like ‘hello’directly, we would need to overload the operator yet again, this time using a char pointer as the argument. Third, notice that we are using the strcat function, which requires us to add the string.h header file. This is one of the reasons we are encouraged to use separate files for class definitions and declarations, to prevent our main file from getting unnecessarily cluttered.

We must also add conditions to make sure that the total length of the combined string is not larger than the length we have accommodated for the character arrays in our class definition (something which has not been done here).

## Safe Array Example

When we use arrays in C++, neither the compiler nor the program will, by default, warn us if we try to access the 20th element in an array of size 5, i.e. we go out of bounds. To add this capability, we could create our own class definitions.

#define **LIMIT** 100  
class safeArray  
{  
 int arr[**LIMIT**];  
public:  
 void putIn(int n, int index)  
 {  
 if (index >= **LIMIT** || index < 0)*//display error message* else arr[index] = n;  
 }  
 int getOut(int index)  
 {  
 if (index >= **LIMIT** || index < 0)*//display error message* else return arr[index];  
 }  
};

C++

The functions in this class check that the index is inside the bounds of the array before attempting to insert or get an element from the array. However, the format we have created is a little bothersome. We will have to use functions like these in the main function:

safeArray s1;  
s1.putIn(5, 0);  
cout<<s1.getOut(0);

C++

Instead of using the two functions to insert or retrieve some element in the array, we could create a single function that returns the element by reference, so that we can do with it as we please.

int& access (int index)  
{  
 if (index >= **LIMIT** || index < 0) *//display error message* else return arr[index];  
}

C++

Remember that when we return something by reference, we are returning the actual element, unlike the copy we return when we return something by value or the address we get when we return something through a pointer. Thus, in the main function, if we write the statement:

s1.access(5) = 10;

C++

which would get us the element at index position 5, and allow us to set its value. We could also use the statement

cout<<s1.access(5);

C++

which would again, get us the element at index position 5, and allow us to print it. The function itself does not do anything at all to the element. It simply allows us to access that element. Notice that we are using a public function that essentially fetches us an element that is private.

The setup we have created looks great, but we can make it even better. We could for example, overload the subscript operator [], which is used to access array elements, to perform the function we just created. That way, we could use statements like

cout<<s1[5];

C++

which come naturally to us, while at the same time maintaining the functionalities of our custom class.

int& operator [] (int index)  
{  
 if (index >= LIMIT || index < 0) *//display error message* else return arr[index];  
}

C++

When we write the statement

s1[5];

C++

the compiler goes to the overloaded function of the subscript operator for the s1 object, taking 5 as an argument for that function.

This will even work if we created an array of objects like this:

safeArray s1[10];

C++

and then tried to access an element like this:

s1[1][5] = 10;

C++

This would work like a normal 2D array. It would go to the first element of the array we created in the main function, which is a single object, and then go to the fifth element of the array we created for that object in the class definition and return that element to us.

## Data Conversion

When we write a statement like this:

int a = 3.5;

C++

the float value is automatically converted by the compiler into an integer. There are many more automatic conversions like this one, that are handled automatically by the compiler using built in routines. These conversions are said to be implicit, since they are not apparent. We can even force a conversion by using the cast operator, though this would use the same built-in routines. Such a conversion would be explicit.

int a = static\_cast<int>3.5;

C++

Now consider objects we create. Say we want to convert the value of the counter integer inside our Counter class to a character. We could first retrieve value, and then convert it, but it would be much better if we could do something like this:

char a = static\_cast<char>(c1);

C++

where c1 is an object of the Counter class. Unfortunately, the compiler has no clue what to do here, since we did not tell it how to handle conversion requests for the class we created. In order to allow this capability, we need to create a conversion operator in the class definition:

operator char() const  
{  
 return static\_cast<char>(Counter);  
}

C++

This is a fairly simple use though. We could have skipped this function entirely if we had access to the counter integer (which we do not since it is private). However, this same syntax can be used for more complicated conversions. For example, if we wanted to convert the data from an object d1 of the Distance class into a float meters, we could perform the calculations inside a function like this one and then call the function like this:

float meters = static\_cast<float>(d1);

C++

or simply

float meters = d1;

C++

On the flip side, if we wanted to convert a basic data type into a user-defined type, we would need to create a conversion constructor in the class definition:

Counter (char a)  
{  
 counter = static\_cast<int>(a);  
}

C++

Now, whenever we create an object of the Counter class using a single character as an argument, or when we assign the object a character value, this function is called and the input is converted to an integer and stored.

Counter c1('a');  
cout<<c1.getCounter(); *//prints 97*c1 = 'b';  
cout<<c1.getCounter(); *//prints 98*

C++

Finally, consider two objects from different classes. Suppose we have two classes, time12 and time24 that record time in 12-hour and 24-hour formats respectively. In either class, we could need to create both a conversion operator and a conversion constructor. Since both the operator and the constructor functions have the name of the other class in their definitions, the other class needs to be declared first. Say we put the functions in the time12 class.

#include<iostream>  
using namespace std;  
  
class time24  
{  
public:  
 int hour;  
};

class time12  
{  
public:  
 int hour;  
 bool noon;  
 operator time24() const *//conversion operator* {  
 time24 temp;  
 if (noon) temp.hour = hour+12; *//if past noon, add 12* else temp.hour = hour;  
 return temp;  
 }

time12 (): hour(0) {}; *//zero argument constructor* time12 (time24 temp) *//conversion constructor* {  
 if (temp.hour > 12) *//if past noon, set noon to true* {  
 hour = temp.hour-12;  
 noon = true;  
 }  
  
 else  
 {  
 hour = temp.hour;  
 noon = false;  
 }  
 }  
};  
  
int main()  
{  
 time12 obj1;  
 obj1.hour = 3;  
 obj1.noon = true; *//3pm* time24 obj2 = obj1; *//convert time12 obj to time24 obj* cout<<obj2.hour<<endl; *//prints out 15* time12 obj3(obj2); *//convert time24 obj to time12 obj* cout<<obj3.hour; *//prints 3 P.M.* if (obj3.noon) cout<<" P.M."<<endl;  
 else cout<<" A.M."<<endl;

obj2.hour = 9;  
  
 obj3 = obj2; *//convert time24 obj to time12 obj* cout<<obj3.hour; *//prints 9 A.M.* if (obj3.noon) cout<<" P.M."<<endl;  
 else cout<<" A.M."<<endl;  
}

C++