**C++ Operator Overloading Guidelines**

One of the nice features of C++ is that you can give special meanings to operators, when they are used with user-defined classes. This is called *operator overloading*. You can implement C++ operator overloads by providing special member-functions on your classes that follow a particular naming convention. For example, to overload the + operator for your class, you would provide a member-function named operator+ on your class.

The following set of operators is commonly overloaded for user-defined classes:

* = (assignment operator)
* + - \* (binary arithmetic operators)
* += -= \*= (compound assignment operators)
* == != (comparison operators)

Here are some guidelines for implementing these operators. These guidelines are very important to follow, so definitely get in the habit early.

**Assignment Operator =**

The assignment operator has a signature like this:

class MyClass {

public:

...

MyClass & operator=(const MyClass &rhs);

...

}

MyClass a, b;

...

b = a; // Same as b.operator=(a);

Notice that the = operator takes a const-reference to the right hand side of the assignment. The reason for this should be obvious, since we don't want to change that value; we only want to change what's on the left hand side.

Also, you will notice that a reference is returned by the assignment operator. This is to allow **operator chaining**. You typically see it with primitive types, like this:

int a, b, c, d, e;

a = b = c = d = e = 42;

This is interpreted by the compiler as:

a = (b = (c = (d = (e = 42))));

In other words, assignment is **right-associative**. The last assignment operation is evaluated first, and is propagated leftward through the series of assignments. Specifically:

* e = 42 assigns 42 to e, then returns e as the result
* The value of e is then assigned to d, and then d is returned as the result
* The value of d is then assigned to c, and then c is returned as the result
* etc.

Now, in order to support operator chaining, the assignment operator must return some value. The value that should be returned is a reference to the *left-hand side* of the assignment.

Notice that the returned reference is *not* declared const. This can be a bit confusing, because it allows you to write crazy stuff like this:

MyClass a, b, c;

...

(a = b) = c; // What??

At first glance, you might want to prevent situations like this, by having operator= return a const reference. However, *statements like this will work with primitive types.* And, even worse, some tools actually rely on this behavior. Therefore, it is important to return a **non-const** reference from your operator=. The rule of thumb is, "If it's good enough for ints, it's good enough for user-defined data-types."

So, for the hypothetical MyClass assignment operator, you would do something like this:

// Take a const-reference to the right-hand side of the assignment.

// Return a non-const reference to the left-hand side.

MyClass& MyClass::operator=(const MyClass &rhs) {

... // Do the assignment operation!

return \*this; // Return a reference to myself.

}

Remember, this is a pointer to the object that the member function is being called on. Since a = b is treated as a.operator=(b), you can see why it makes sense to return the object that the function is called on; object a *is* the left-hand side.

But, the member function needs to return a reference to the object, not a pointer to the object. So, it returns \*this, which returns what this points at (i.e. the object), not the pointer itself. (In C++, instances are turned into references, and vice versa, pretty much automatically, so even though \*this is an instance, C++ implicitly converts it into a reference to the instance.)

Now, one more **very important** point about the assignment operator:

**YOU MUST CHECK FOR SELF-ASSIGNMENT!**

This is especially important when your class does its own memory allocation. Here is why: The typical sequence of operations within an assignment operator is usually something like this:

MyClass& MyClass::operator=(const MyClass &rhs) {

// 1. Deallocate any memory that MyClass is using internally

// 2. Allocate some memory to hold the contents of rhs

// 3. Copy the values from rhs into this instance

// 4. Return \*this

}

Now, what happens when you do something like this:

MyClass mc;

...

mc = mc; // BLAMMO.

You can hopefully see that this would wreak havoc on your program. Because mc is on the left-hand side *and* on the right-hand side, the first thing that happens is that mc releases any memory it holds internally. But, this is where the values were going to be copied from, since mc is also on the right-hand side! So, you can see that this completely messes up the rest of the assignment operator's internals.

The easy way to avoid this is to **CHECK FOR SELF-ASSIGNMENT.** There are many ways to answer the question, "Are these two instances the same?" But, for our purposes, just compare the two objects' addresses. If they are the same, then don't do assignment. If they are different, then do the assignment.

So, the correct and safe version of the MyClass assignment operator would be this:

MyClass& MyClass::operator=(const MyClass &rhs) {

// Check for self-assignment!

if (this == &rhs) // Same object?

return \*this; // Yes, so skip assignment, and just return \*this.

... // Deallocate, allocate new space, copy values...

return \*this;

}

Or, you can simplify this a bit by doing:

MyClass& MyClass::operator=(const MyClass &rhs) {

// Only do assignment if RHS is a different object from this.

if (this != &rhs) {

... // Deallocate, allocate new space, copy values...

}

return \*this;

}

Remember that in the comparison, this is a pointer to the object being called, and &rhs is a pointer to the object being passed in as the argument. So, you can see that we avoid the dangers of self-assignment with this check.

In summary, the guidelines for the assignment operator are:

1. Take a const-reference for the argument (the right-hand side of the assignment).
2. Return a reference to the left-hand side, to support safe and reasonable operator chaining. (Do this by returning \*this.)
3. Check for self-assignment, by comparing the pointers (this to &rhs).

**Compound Assignment Operators += -= \*=**

I discuss these before the arithmetic operators for a very specific reason, but we will get to that in a moment. The important point is that these are *destructive* operators, because they update or replace the values on the left-hand side of the assignment. So, you write:

MyClass a, b;

...

a += b; // Same as a.operator+=(b)

In this case, the values within a are *modified* by the += operator.

How those values are modified isn't very important - obviously, what MyClass represents will dictate what these operators mean.

The member function signature for such an operator should be like this:

MyClass & MyClass::operator+=(const MyClass &rhs) {

...

}

We have already covered the reason why rhs is a const-reference. And, the implementation of such an operation should also be straightforward.

But, you will notice that the operator returns a MyClass-reference, and a non-const one at that. This is so you can do things like this:

MyClass mc;

...

(mc += 5) += 3;

Don't ask me why somebody would want to do this, but just like the normal assignment operator, this is allowed by the primitive data types. Our user-defined datatypes should match the same general characteristics of the primitive data types when it comes to operators, to make sure that everything works as expected.

This is very straightforward to do. Just write your compound assignment operator implementation, and return \*this at the end, just like for the regular assignment operator. So, you would end up with something like this:

MyClass & MyClass::operator+=(const MyClass &rhs) {

... // Do the compound assignment work.

return \*this;

}

As one last note, *in general* you should beware of self-assignment with compound assignment operators as well. Fortunately, none of the C++ track's labs require you to worry about this, but you should always give it some thought when you are working on your own classes.

**Binary Arithmetic Operators + - \***

The binary arithmetic operators are interesting because they don't modify either operand - they actually return a new value from the two arguments. You might think this is going to be an annoying bit of extra work, but here is the secret:

**Define your binary arithmetic operators using your compound assignment operators.**

There, I just saved you a bunch of time on your homeworks.

So, you have implemented your += operator, and now you want to implement the + operator. The function signature should be like this:

// Add this instance's value to other, and return a new instance

// with the result.

const MyClass MyClass::operator+(const MyClass &other) const {

MyClass result = \*this; // Make a copy of myself. Same as MyClass result(\*this);

result += other; // Use += to add other to the copy.

return result; // All done!

}

Simple!

Actually, this explicitly spells out all of the steps, and if you want, you *can* combine them all into a single statement, like so:

// Add this instance's value to other, and return a new instance

// with the result.

const MyClass MyClass::operator+(const MyClass &other) const {

return MyClass(\*this) += other;

}

This creates an unnamed instance of MyClass, which is a *copy* of \*this. Then, the += operator is called on the temporary value, and then returns it.

If that last statement doesn't make sense to you yet, then stick with the other way, which spells out all of the steps. But, if you understand exactly what is going on, then you can use that approach.

You will notice that the + operator returns a const instance, *not* a const reference. This is so that people can't write strange statements like this:

MyClass a, b, c;

...

(a + b) = c; // Wuh...?

This statement would basically do nothing, but if the + operator returns a non-const value, it *will* compile! So, we want to return a const instance, so that such madness will not even be allowed to compile.

To summarize, the guidelines for the binary arithmetic operators are:

1. Implement the compound assignment operators from scratch, and then define the binary arithmetic operators in terms of the corresponding compound assignment operators.
2. Return a const instance, to prevent worthless and confusing assignment operations that shouldn't be allowed.

**Comparison Operators == and !=**

The comparison operators are very simple. Define == first, using a function signature like this:

bool MyClass::operator==(const MyClass &other) const {

... // Compare the values, and return a bool result.

}

The internals are very obvious and straightforward, and the bool return-value is also very obvious.

The important point here is that the != operator can also be defined in terms of the == operator, and you should do this to save effort. You can do something like this:

bool MyClass::operator!=(const MyClass &other) const {

return !(\*this == other);

}

That way you get to reuse the hard work you did on implementing your == operator. Also, your code is far less likely to exhibit inconsistencies between == and !=, since one is implemented in terms of the other.

**Classes (II)**

**Overloading operators**

C++ incorporates the option to use standard operators to perform operations with classes in addition to with fundamental types. For example:

|  |  |
| --- | --- |
| 1 2 | int a, b, c;  a = b + c; |

This is obviously valid code in C++, since the different variables of the addition are all fundamental types. Nevertheless, it is not so obvious that we could perform an operation similar to the following one:

|  |  |
| --- | --- |
| 1 2 3 4 5 | struct {  string product;  float price;  } a, b, c;  a = b + c; |

In fact, this will cause a compilation error, since we have not defined the behavior our class should have with addition operations. However, thanks to the C++ feature to overload operators, we can design classes able to perform operations using standard operators. Here is a list of all the operators that can be overloaded:

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

To overload an operator in order to use it with classes we declare *operator functions*, which are regular functions whose names are the operator keyword followed by the operator sign that we want to overload. The format is:  
  
type operator sign (parameters) { /\*...\*/ }  
  
Here you have an example that overloads the addition operator (+). We are going to create a class to store bidimensional vectors and then we are going to add two of them: a(3,1) and b(1,2). The addition of two bidimensional vectors is an operation as simple as adding the two x coordinates to obtain the resulting x coordinate and adding the two y coordinates to obtain the resulting y. In this case the result will be (3+1,1+2) = (4,3).

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 | // vectors: overloading operators example  #include <iostream>  using namespace std;  class CVector {  public:  int x,y;  CVector () {};  CVector (int,int);  CVector operator + (CVector);  };  CVector::CVector (int a, int b) {  x = a;  y = b;  }  CVector CVector::operator+ (CVector param) {  CVector temp;  temp.x = x + param.x;  temp.y = y + param.y;  return (temp);  }  int main () {  CVector a (3,1);  CVector b (1,2);  CVector c;  c = a + b;  cout << c.x << "," << c.y;  return 0;  } | 4,3 |

It may be a little confusing to see so many times the CVector identifier. But, consider that some of them refer to the class name (type) CVector and some others are functions with that name (constructors must have the same name as the class). Do not confuse them:

|  |  |
| --- | --- |
| 1 2 | CVector (int, int); // function name CVector (constructor)  CVector operator+ (CVector); // function returns a CVector |

The function operator+ of class CVector is the one that is in charge of overloading the addition operator (+). This function can be called either implicitly using the operator, or explicitly using the function name:

|  |  |
| --- | --- |
| 1 2 | c = a + b;  c = a.operator+ (b); |

Both expressions are equivalent.  
  
Notice also that we have included the empty constructor (without parameters) and we have defined it with an empty block:

|  |  |
| --- | --- |
|  | CVector () { }; |

This is necessary, since we have explicitly declared another constructor:

|  |  |
| --- | --- |
|  | CVector (int, int); |

And when we explicitly declare any constructor, with any number of parameters, the default constructor with no parameters that the compiler can declare automatically is not declared, so we need to declare it ourselves in order to be able to construct objects of this type without parameters. Otherwise, the declaration:

|  |  |
| --- | --- |
|  | CVector c; |

included in main() would not have been valid.  
  
Anyway, I have to warn you that an empty block is a bad implementation for a constructor, since it does not fulfill the minimum functionality that is generally expected from a constructor, which is the initialization of all the member variables in its class. In our case this constructor leaves the variables x and y undefined. Therefore, a more advisable definition would have been something similar to this:

|  |  |
| --- | --- |
|  | CVector () { x=0; y=0; }; |

which in order to simplify and show only the point of the code I have not included in the example.  
  
As well as a class includes a default constructor and a copy constructor even if they are not declared, it also includes a default definition for the assignment operator (=) with the class itself as parameter. The behavior which is defined by default is to copy the whole content of the data members of the object passed as argument (the one at the right side of the sign) to the one at the left side:

|  |  |
| --- | --- |
| 1 2 3 | CVector d (2,3);  CVector e;  e = d; // copy assignment operator |

The copy assignment operator function is the only operator member function implemented by default. Of course, you can redefine it to any other functionality that you want, like for example, copy only certain class members or perform additional initialization procedures.  
  
The overload of operators does not force its operation to bear a relation to the mathematical or usual meaning of the operator, although it is recommended. For example, the code may not be very intuitive if you use operator + to subtract two classes or operator== to fill with zeros a class, although it is perfectly possible to do so.  
  
Although the prototype of a function operator+ can seem obvious since it takes what is at the right side of the operator as the parameter for the operator member function of the object at its left side, other operators may not be so obvious. Here you have a table with a summary on how the different operator functions have to be declared (replace @ by the operator in each case):

|  |  |  |  |
| --- | --- | --- | --- |
| **Expression** | **Operator** | **Member function** | **Global function** |
| @a | + - \* & ! ~ ++ -- | A::operator@() | operator@(A) |
| a@ | ++ -- | A::operator@(int) | operator@(A,int) |
| a@b | + - \* / % ^ & | < > == != <= >= << >> && || , | A::operator@ (B) | operator@(A,B) |
| a@b | = += -= \*= /= %= ^= &= |= <<= >>= [] | A::operator@ (B) | - |
| a(b, c...) | () | A::operator() (B, C...) | - |
| a->x | -> | A::operator->() | - |

Where a is an object of class A, b is an object of class B and c is an object of class C.  
  
You can see in this panel that there are two ways to overload some class operators: as a member function and as a global function. Its use is indistinct, nevertheless I remind you that functions that are not members of a class cannot access the private or protected members of that class unless the global function is its friend (friendship is explained later).

**The keyword this**

The keyword this represents a pointer to the object whose member function is being executed. It is a pointer to the object itself.  
  
One of its uses can be to check if a parameter passed to a member function is the object itself. For example,

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 | // this  #include <iostream>  using namespace std;  class CDummy {  public:  int isitme (CDummy& param);  };  int CDummy::isitme (CDummy& param)  {  if (&param == this) return true;  else return false;  }  int main () {  CDummy a;  CDummy\* b = &a;  if ( b->isitme(a) )  cout << "yes, &a is b";  return 0;  } | yes, &a is b |

It is also frequently used in operator= member functions that return objects by reference (avoiding the use of temporary objects). Following with the vector's examples seen before we could have written an operator= function similar to this one:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 | CVector& CVector::operator= (const CVector& param)  {  x=param.x;  y=param.y;  return \*this;  } |

In fact this function is very similar to the code that the compiler generates implicitly for this class if we do not include an operator= member function to copy objects of this class.

**Static members**

A class can contain *static* members, either data or functions.  
  
Static data members of a class are also known as "class variables", because there is only one unique value for all the objects of that same class. Their content is not different from one object of this class to another.   
  
For example, it may be used for a variable within a class that can contain a counter with the number of objects of that class that are currently allocated, as in the following example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 | // static members in classes  #include <iostream>  using namespace std;  class CDummy {  public:  static int n;  CDummy () { n++; };  ~CDummy () { n--; };  };  int CDummy::n=0;  int main () {  CDummy a;  CDummy b[5];  CDummy \* c = new CDummy;  cout << a.n << endl;  delete c;  cout << CDummy::n << endl;  return 0;  } | 7  6 |

In fact, static members have the same properties as global variables but they enjoy class scope. For that reason, and to avoid them to be declared several times, we can only include the prototype (its declaration) in the class declaration but not its definition (its initialization). In order to initialize a static data-member we must include a formal definition outside the class, in the global scope, as in the previous example:

|  |  |
| --- | --- |
|  | int CDummy::n=0; |

Because it is a unique variable value for all the objects of the same class, it can be referred to as a member of any object of that class or even directly by the class name (of course this is only valid for static members):

|  |  |
| --- | --- |
| 1 2 | cout << a.n;  cout << CDummy::n; |

These two calls included in the previous example are referring to the same variable: the static variable n within class CDummy shared by all objects of this class.  
  
Once again, I remind you that in fact it is a global variable. The only difference is its name and possible access restrictions outside its class.  
  
Just as we may include static data within a class, we can also include static functions. They represent the same: they are global functions that are called as if they were object members of a given class. They can only refer to static data, in no case to non-static members of the class, as well as they do not allow the use of the keyword this, since it makes reference to an object pointer and these functions in fact are not members of any object but direct members of the class.