The ABCs of Writing C++ Classes

Constructors, Destructors, and Assignment Operators

by**G. Bowden Wise**

As you develop your own classes, you will ask yourself many questions. Do I need a copy constructor? Are default arguments necessary in my constructors? How do I provide type conversion for my class? Do I need an equality operator? How do I implement a postfix operator? Should I return a reference or an object? How do I stream this object to disk?

Even experienced C++ programmers ask themselves these questions over and over and even make mistakes that make their code inefficient. Many books [2,1,5,4] and columns, such as Andrew Koenig's C ++ column [3] found in *JOOP*, have been written that discuss the many pitfalls to avoid and provide guidelines and insights to use when implementing C++ classes. Becoming a better C++ programmer requires that you learn some these subtle dangers and insights.

For the next several issues of this column I will present some of the useful guidelines taken from these masters of the C++ language and add a few of my own. I hope that these guidelines will prove useful to you as you implement your own classes. We will begin with the special member functions of a class: constructors, destructors, and assignment operators.

Special Member Functions

Classes contain both data and functions as members. Most functions provide an interface to the object to allow users to apply operations to the class. Other functions have a special meaning, since they play a special role during the lifetime of an object. The special member functions of a class are the constructors, destructor, and assignment operator.

The Lifetime of An Object

Objects are introduced into your program through declarations. Other times, the compiler may generate code to create objects for its own purposes. You assign objects a name when they are declared. These named objects can only be used in *scopes* in which they are defined. A scope is a region of program text. Objects can only be used within scopes in which they are defined.

There are three kinds of scope: *local scope*, which include objects defined in blocks of code; *file scope*, which includes objects in a file, but outside of all blocks and classes; and *class scope*, which includes objects declared as data members of a class. Objects of file scope are also considered to be *global*.

The lifetime of an object is determined by its *storage class*:

- *Automatic*. Objects declared within a block are considered automatic. Automatic objects are created and initialized each time the control flow of the program reaches its definition and destroyed upon exit from the block.
- *Static*. Objects prefixed with the keyword `static` are static and exist and retain their values during the entire execution of the program. Global objects are also static. Static objects are created and initialized before the first use of any function in a particular translation unit. Static objects are by default automatically initialized to zero.

C++ provides mechanisms for ensuring that your objects are properly initialized before they are used. As your objects go in and out of scope, memory is allotted for them and that memory is then initialized. C++ provides a special member function for the initialization of an object, called the *constructor*.

A constructor is called whenever an object is created. Objects can be created: as a global variable, as a local variable, through explicit use of the `new` operator, through an explicit call of a constructor, or as a temporary object. Constructors are also called when an object is created as part of another object.

Similarly, when your object goes out of scope, the memory used by the object must be reclaimed. C++ provides a special member function, called the *destructor*, that is called whenever your object is destroyed so that you may perform any clean-up processing, such as freeing memory or other system resources obtained by the object.

C++ also provides two other special functions that play a special role. Whenever an object must be copied, its *copy constructor* is invoked. Finally, whenever an object is assigned a value, its *assignment operator* is invoked.

Constructors

The *constructor* is responsible for turning the raw memory allotted to an object into a usable object. Constructors come in many forms. There are default constructors, copy constructors, and other constructors that take different arguments.

Default constructor

The *default constructor* is a constructor that takes no arguments. The default constructor for a class `X` has the form `X::X()`. A constructor which has all *default arguments*, `X::X(const int x=0)`, for example, is also a default constructor, since it can be called with no arguments.

Default constructors allow objects to be created without passing any parameters to the constructor. For example, the declaration

```
   String s;
```

results in a string `s` that does not yet have a value; it is an empty string.

The default constructor usually creates an object that represents a ``null'' instance of the particular type the class denotes. The default constructor for a complex number might result in an object with value zero, while the default constructor for a linked list might would result in an empty list.

Guideline 1. *Provide a default constructor for your class. Use default arguments to avoid having a separate default constructor that has no arguments.*

Often you will allow users of your classes to pass arguments to the constructor. Rather than provide a separate default constructor that takes no arguments, it is better to provide a constructor with default arguments that can serve as a default constructor or as a constructor that takes the arguments it specifies. This makes your code more compact and leads to more code reuse.

For example, the two constructors of a `String`, `String()` and `String(const char* str)`, can be combined into a single constructor that has a default argument: `String(const char* str=0)`.

There can only be **one** default constructor, so don't add default arguments to all the arguments of every constructor - your compiler will likely complain.

When defining your classes, it may appear that you cannot have a default constructor because you need certain data members to be initialized at creation. However, if you provide a way to ensure that these data members are later provided then you can still have a default constructor. For example, if you have used the C++ iostream library, you know that you can declare an output file without giving the file name:

```
   ofstream out;
```

The file name is later provided during the open call:

```
   out.open("outfile");
```

Be careful when implementing templates. Sometimes if you aren't careful how you implement the code, you can force your users to require that their types have a default constructor. If you place

```
   T element;
```

in one of your template functions, you are forcing your users to use types that have a default constructor.

Guideline 2. *Use default arguments in all member functions where appropriate.*

Default arguments are not restricted to constructors, they can be used by any member function. Provide appropriate default values wherever appropriate.

Copy constructor

A *copy constructor* is a special constructor that can be called to copy an object. The copy constructor for class `X` has the form `X::X(const X&)`. In the code fragment below, `s2` is created by calling `String`'s copy constructor to copy `s1`:

```
   String s1("Hello, world!");
   String s2 (s1);
```

The copy constructor is called more often behind the scenes whenever the compiler needs a copy of an object. These objects, appropriately called *temporaries*, are created and destroyed as they are needed by the compiler. The most common place this occurs is during function calls, so that the semantics of call-by-value can be preserved. For example, here is a function that takes a `String` argument:

```
   void DisplayError (const String s);
```

Whenever `DisplayError()` is called, the compiler generates a call to the copy constructor for `String` to create a temporary for parameter `s`. The temporary is then passed to the function.

Guideline 3. *Always provide a copy constructor for your classes. Do not let the compiler generate it for you. If your class has pointer data members, you* ***must*** *provide the copy constructor.*

If you do not provide a copy constructor, the compiler will generate one for you automatically. This generated copy constructor simply performs a member-wise assignment of all of the data members of a class. This is fine for a class that does not contain any pointer variables. It is a good idea to get in the habit of always providing the copy constructor for your classes.

Guidelines for Implementing Constructors

Constructors are called frequently, not only by you when you declare objects, but also when the compiler creates temporaries. It is, therefore, important for the constructors to be compact and as efficient as possible. Below are a few guidelines to consider when implementing constructors.

Guideline 4. _ *When initializing data members from the constructor, use initialization rather than assignment. This will make your code more efficient and avoid extra calls to the constructors.*

Suppose you have a `Shape` class with data members: `_center` of type `Point` and `_color` of type `int`. When implementing the constructor, you might be tempted to use assignment, to initialize the data members:

```
Shape::Shape (const Point center,
              const int   color)
{
   _center = center;
   _color  = color;
}
```

However, this results in an extra call to the constructor for `Point` and makes the code execute slower. To see why, let's look at how objects are constructed. There are two phases to object construction:

1. Data members are initialized in the order of their declaration in the class. This is called *member initialization*.
2. The body of the constructor is executed.

Also, for derived classes, these two steps are performed on the base classes first.

In the `Shape` constructor, `_center` is constructed during Step 1, when the default constructor for `Point` is invoked. Then in Step 2, when the body of the constructor executes the assignment operator is invoked which changes the value of `_center`.

Note that `_center` will always be initialized before the body of the constructor is ever entered. However, you can control which constructor is called for `_center` in the initialization list of the `Shape` constructor:

```
Shape::Shape (const Pointer center,
              const int     color)
: _center(center)
, _color (color)
{
}
```

Specifying `_center(center)` in the initialization list, tells the compiler to call the copy constructor rather than the default constructor during member initialization. Since `_center` is properly initialized, the assignment is no longer needed in the body of the constructor.

When implementing your constructors, try to initialize all of your data members using this technique. In most cases, all can be initialized this way, so there will be no need for any code in the body of the constructor. This makes your code much more readable and maintainable.

Actually, built-in types, such as `int`, do not have constructors. Therefore, it makes no difference whether you use assignment or initialization for variables with built-in types. But, the code is more manageable and easier to read if all data members are initialized the same way.

Guideline 5. *Pay attention to the order of member initialization. If you use a data member,* `d1` *to initialize another data member* `d2`*, make sure* `d1` *is in fact initialized before initializing* `d2`*.*

Consider a `StringHandle` class which declares data members in in the following order:

```
String _str;
int    _handle;
```

C++ language rules tell us that the members of a class are initialized in the order they appear in the class declaration, **not** in the order they appear in the initialization list of the constructor (e.g., `_str` will be initialized before `_handle`).

The following constructor will compile, but will result in run-time error:

```
StringHandle::StringHandle (const int h)
: _handle (h),_str(QueryHandle(_handle))
{
}
```

Because `_str` appears first in the class declaration, it will be intilialized before `_handle`. Unfortunately, `_str` uses `_handle` in its construction, which has not yet been constructed. Specifying `_handle` before `_str` in the initializer list doesn't matter.

To fix this problem, we can use the incoming argument instead of the class member:

```
StringHandle::StringHandle (const int h)
```

```
    : _handle(h),_str (QueryStringHandle(h))
    {
    }
```

The reason that the initializer list order is ignored is so that the compiler can ensure that variables are destroyed in the reverse order of their construction. There is no guarantee that the constructor and destructor are implemented in the same source file. However, both the constructor and destructor see the same class declaration, so there is no ambiguity involved by using the declaration to drive the order of initialization.

Guideline 6. *Reference data members and const objects* ***must*** *be initialized using the initializer syntax.*

Consider the class, `SensorData`, which manages sensor data in a laboratory data. It can work with a variety of different sensors, so its constructor takes a reference to an existing sensor. Some global data available to all sensors is passed as a constant.

References must point to another already existing object and, therefore, must be initialized to point at the aliased object during initialization. A reference can only have one assignment for its lifetime. Similarly, constant objects can never be assigned a value, so they must also be initialized using an initializer. Here is a possible implementation of the constructor:

```
    SensorData::SensorData (Sensor& aSensor
                            const GlobalData data)
    : _rSensor(aSensor), _global (data)
    {
    }
```

Guideline 7. *Make the constructor as compact as possible to reduce overhead and to minimize errors during construction.*

The constructor is called each time an object is created. Lots of times this occurs without your control, such as when the compiler creates temporaries. Initialization of an object should not take a very long time. Only do what is minimally necessary during construction. Reducing the amount of processing you perform during construction also reduces the chance of an error occurring.

Guideline 8. _ *Remember to invoke base class constructors from derived class constructors.*

Constructors (as well as destructors and assignment operators) are not inherited by derived classes. During member initialization, just before the constructor body is executed, constructors are called for the data members and base classes of the class. As we saw in Guideline **4**, the compiler uses the initializer list to determine which constructor to call. If you do not call the base class constructor from the initializer list of a derived class, the compiler will generate a call to the default constructor of the base class. If the base class does not have a default constructor, the compiler will complain.

It is a good idea to initialize base classes and your data members from the initializer list of derived classes. I tend to put my calls to base class constructors before initializing the data members.

Suppose we decided to derive a class `Triangle` from `Shape`:

```
class Triangle : public Shape {
public:
   Triangle (const Point center,
             const int color,
             const Point v1,
             const Point v2,
             const Point v3);
private:
   Point _v1, _v2, _v3;
};
```

We call the `Shape` constructor from the initialization list of `Triangle`'s constructor:

```
Triangle::Triangle (const Point center,
                    const int color,
                    const Point v1,
                    const Point v2,
                    const Point v3)
: Shape (center, color)
, _v1(v1), _v2(v2), _v3(v3)
{
}
```

Destructors

Whenever an object goes out of scope it is destroyed and the memory used by that object is reclaimed. Just before the object is destroyed, an object's *destructor* is called to allow any clean-up to be performed. A destructor for class `X` has the form `X::~X()`.

Guideline 9. *The destructor must release any resources obtained by an object during its* ***lifetime****, not just those that were obtained during construction.*

Typically, some data members for an object are allocated during construction. However, some objects may allocate memory or use other resources in response to certain operations performed on the object.

When implementing your destructors, make sure you release **all** of the resources that have been obtained by the object, not just those obtained by the constructors.

Guideline 10. *Always provide a destructor for your classes. Do not let the compiler generate it for you.*

You want to be in complete control of how your class operates. Do not let the compiler generate any of your code.

Guideline 11. *If your class has any possibility of being used as a base class for which other classes are derived from make the destructor virtual. If your class contains at least one virtual member function, make sure the destructor is also virtual.*

Consider a hierarchy of automobiles. All classes in the hierarchy inherit from the base class `Auto`. When a `Ford` object is created:

```
   Ford f;
```

Constructors are called from base to derived, resulting in this sequence of constructor calls: `Auto(); Ford()`. Similarly, when the object `f` is destroyed, constructors are called in reverse order: `Ford(); Auto()`. However, problems result when we create and destroy objects through a base class pointer:

```
   Auto* car1 = new Ford;
   // ...
   delete car1;
```

Since `Auto`'s constructor is not virtual, the `delete` statement calls `Auto::~Auto()` and not `Ford::~Ford()` as desired. Declaring `Auto::~Auto()` virtual solves this problem. Making a destructor virtual ensures that the appropriate destructor is invoked whenever `delete` is applied to a base class pointer. Once the correct constructor is invoked, the destructors for the base classes are invoked in term in the usual order as shown above.

Guidelines for constructors and destructors

There are some guidelines that apply to both constructors and destructors. We discuss these guidelines here.

Guideline 12. *Avoid calls to virtual functions in constructors and destructors.*

You may call other member functions from constructors and destructors. Calling a virtual function will result in a call to the one defined in the constructor's (or destructor's) own class or its bases, but not any function overriding it in a derived class. This will guarantee that unconstructed derived objects will not be accessed during construction or destruction.

Also, calling a pure virtual function directly or indirectly for the object being constructed or destroyed is undefined. The only time it is valid is if you explicitly call the pure virtual function. For example, if class A has pure virtual function `p()`, then: `A::p()` is an explicit call. The reason this is works is because explicit qualification circumvents the virtual calling mechanism.

The Assignment Operator

Whenever you perform an assignment to an object, the class's *assignment operator* is invoked. The assignment operator for a class `X` has one of two forms:

```
X& X::operator= (const X& x);
const X& X::operator= (const X& x);
```

Guideline 13. *The assignment operator must be a member function; it cannot be a friend.*

Some operators are best implemented as member functions, while others are best implemented as friend functions. However, the assignment operator **must** be implemented as a member using one of the two forms shown above.

Guideline 14. *Always return a reference to the object being assigned.*

References are an alias to an already existing object. Whatever operations are applied to the reference are also applied to the aliased object. Recall that references are simply another name for an object (e.g., `printf("%p %p \n",&,&)` produces `0xbffff870 0xbffff870`).

The reasons for returning a reference are twofold:

1. it allows, several assignments to be chained together, as in:

    ```
    Complex w, x, y, z;
    w = x = y = z = Complex(0,0);
    ```

2. and, it is more efficient because a copy of the returned object does not have to be created and returned.

In order for the cascaded assignment to work the return type and the argument of the assignment operator must be of the same type. Hence, we return and pass a reference. Typically, you return `*this`. This enables the return of one assignment to become the parameter argument of the next.

Returning a reference is more efficient than returning an object. When references are passed, the default call-by-value mechanism is overridden. Instead of receiving a copy of the argument the function receives the lvalue of the actual argument. No objects are copied, and a reference takes up less space on the stack than the whole object. Similarly, when a reference is returned instead of an object, the lvalue is returned rather than an object.

Guideline 15. *_15 Always return a const reference to the object, returning a non-const reference may result in bizarre results.*

Returning a reference is the same as returning an lvalue. This is just like having a pointer to an object, since we can modify the object using its reference.

Here is an example. Although assignment is right-associative, we can use parentheses to control the associativity:

```
Complex x, y, z;
(x = y) = z;
```

Writing this in equivalent functional form:

```
(x.operator=(y)).operator=(z);
```

What does this mean? First, `x` is assigned to `y`. But, since `operator=` returns a reference to `x`, so `x` is assigned again! This time to `z`. If we return a `const` reference, we prevent the returned reference from being used to modify an object the referenced object. Return const references to prevent accidental modification of your objects.

There is one case where it is desirable to return a non-const reference. The subscript operator must return a reference so that the indexed object can be assigned to. For example, we might wish to write code that changes a coordinate of a vector:

```
Vector v;
v[1] = -2.0;
```

Rewriting this in its equivalent functional form yields:

```
(v.operator[](1)).operator= (-2.0);
```

For this to work, we have the subscript operator return a reference:

```
float& Vector::operator[] (const int)
{
   return _data[i];
}
```

Because the return value is the lvalue of the indexed element, it can appear as the target of an assignment.

Guideline 16. *Check for assignment to self.*

Disaster can result if a variable is assigned to itself. Consider:

```
X x;
x = x;
```

Suppose class `X` contains pointer data members that are dynamically allocated whenever an object is created. Assignment always modifies an existing object. The dynamic data will, therefore, be released before assigning the new value. If we do not check for assignment to self, the above assignment will delete the existing pointer data, and then try to copy the data that was previously deleted!

Always check for assignment to self, as shown in this example:

```
const X& X::operator= (const X& rhs)
{
   if ( &rhs != this )
   {
      // Assign data members here
   }
   return *this;
}
```

Guideline 17. *The assignment operator is the only operator that is not inherited. Make sure that base class data are also updated during assignment.*

The assignment operator of a derived class not only must assign its own data members, but must also ensure that the data members of the base classes are also updated. These base data members might be private and unaccessible to the derived class. Therefore, in order to ensure the base class data is also updated during assignment, the base class assignment operator must be invoked by the derived class assignment operator.

Suppose that class `Base` is the base class of `Derived` Whenever an object of type `Derived` is assigned, a call to the `Base` class assignment operator must be invoked to ensure that the data members of the base part of the object are also updated. Therefore, `Derived`'s assignment operator may invoke `Base`'s assignment by an explicit call (`d` is the argument of the operator):

```
   Base::operator= (d);
```

or through an actual assignment:

```
   ((Base &) *this) = d;
```

This odd syntax casts `*this` into a reference to a `Base` and then makes an assignment to the result of the cast, resulting in an assignment only to the base part of the `Derived` object. It is very important that you cast to a reference to a `Base` object. If you were to cast to a `Base` object, instead of a reference, the copy constructor for `Base` would be called, resulting in a new object. The assignment would be applied to the new object and not to `*this`, leaving the base part of `*this` unaffected by the assignment!

Guideline 18. *Always implement the assignment operator for your class; do not let the compiler generate the default assignment operator.*

The compiler will generate a default assignment operator for your class if you do not provide one. In order to be in complete control of how your class operates, always provide an assignment operator.

Guideline 19. *If your class has pointer data, you* ***must*** *provide an assignment operator. If writing an assignment operator, you must also write a copy constructor.*

The generated assignment operator performs member-wise assignment on any data members of your class. For pointer variables, we almost always do not want this because the data members of the copy will point to the same data as the copied object! Worse, if one of the objects is destroyed, the data is destroyed with it. A run-time error will occur the next time the remaining object tries to access the now non-existent data.

Guideline 20. *Place the common code used by the assignment operator and the copy constructor in a separate function and have each one call the function. This will make your code more compact and avoid duplication.*

A `String` class must copy a character string during the copy constructor and during an assignment. If we place this common code into a private member function

```
   void CopyString(const char* ptr);
```

then both the copy constructor and assignment operator may call this routine to perform the copy rather than duplicate this code in each.

Next time we will look at some guidelines for implementing operators for your classes. Until then, I urge you to take a look at the classes you have already written and see if your constructors, destructors, and assignment operators are behaving correctly. The guidelines presented here may help you write code that is more efficient.

References

1
COPLIEN, J. O. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, Reading, Mass., 1992.

2
ELLIS, M. A., AND STROUSTRUP, B. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Mass., 1990.

3
Journal of Object-Oriented Programming. SIGS Publications, New York.

4
MURRAY, R. B. *C++ Strategies and Tactics*. Addison-Wesley, Reading, Mass., 1993.

5
MYERS, S. *Effective C++*. Addison-Wesley, Reading, Mass., 1992.