V22.0474-001 Software Engineering
Spring 2008

Lecture 14: Effective C++ I

Clark Barrett, New York University

Outline

Effective C++

- Constructors, Destructors, and Assignment Operators
- Classes and Functions: Design and Declaration

Sources for today's lecture:

Meyers, Scott. *Effective C++*, Second Edition.

Constructors, Destructors, and Assignment Operators

Almost every class has constructors, destructors, and assignment operators.
These are the most fundamental class operations, so it's critical to get them right.

We will cover several important C++ design principles regarding these operations.

Constructors, Destructors, and Assignment Operators

Almost every class has constructors, destructors, and assignment operators. These are the most fundamental class operations, so it's critical to get them right.

We will cover several important C++ design principles regarding these operations.

Declare a copy constructor and an assignment operator for classes with dynamically allocated memory (Item 11).

- *What is wrong with the following class definition?*

Constructors, Destructors, and Assignment Operators

```
class String {
public:
  String(const char *value);
  ~String() { delete [] data; \}
  ...   // no copy constructor or operator=

private:
  char *data;
};

String::String(const char *value)
{
  if (value) {
    data = new char[strlen(value) + 1];
    strcpy(data, value);
  }
  else {
    data = new char[1];
    *data = '\0';
  }
}
```

Constructors, Destructors, and Assignment Operators

Consider the following code:

```
String a("Hello");
{
  String b("World");
  b = a;
}
String c = a;
```

Why is this bad?

Constructors, Destructors, and Assignment Operators

Consider the following code:

```
String a("Hello");
{
  String b("World");
  b = a;
}
String c = a;
```

Why is this bad?

- When `b = a` is executed, the string `"World"` is lost.
- When `b` goes out of scope, the string `"Hello"` is deleted, even though `a` still contains this string.
- When `c` is constructed (with an automatically-generated copy constructor), it will also contain the deleted string.
- The `"Hello"` string will get deleted 3 times (the result of deleting an already-deleted pointer is undefined).

Constructors, Destructors, and Assignment Operators

The principle here is that C++ assumes that classes can be assigned and copied.

If you don't declare an assignment or copy constructor, C++ will generate one for you. For classes with dynamically allocated memory, these automatic methods are almost certainly wrong.

Sometimes, it doesn't make sense to copy or assign objects.

What if you want to explicitly disallow assignment or copying for a particular class?

Constructors, Destructors, and Assignment Operators

The principle here is that C++ assumes that classes can be assigned and copied.

If you don't declare an assignment or copy constructor, C++ will generate one for you. For classes with dynamically allocated memory, these automatic methods are almost certainly wrong.

Sometimes, it doesn't make sense to copy or assign objects.

What if you want to explicitly disallow assignment or copying for a particular class?

Declare the assignment and copy constructors to be private:

```
class Array {
private:
  Array& operator=(const Array& rhs);
  ...
};
```

Constructors, Destructors, and Assignment Operators

Prefer initialization to assignment in constructors (Item 12).

Consider the following class definition:

```
template<class T>
class NamedPtr {
public:
  NamedPtr(const string& initName, T *initPtr);
  ...
private:
  string name;
  T *ptr;
};
```

Which is the better constructor definition?:

```
template<class T>
NamedPtr<T>::NamedPtr(const string& initName, T *initPtr)
: name(initName), ptr(initPtr) {}
```

```
template<class T>
NamedPtr<T>::NamedPtr(const string& initName, T *initPtr)
{ name = initName; ptr = initPtr; }
```

Constructors, Destructors, and Assignment Operators

Initialization is preferable to assignment for two main reasons:

- Some class members *must* be initialized, namely `const` and reference members.
- It is always equally or more efficient to use initialization.

Constructors, Destructors, and Assignment Operators

List members in an initialization list in the order in which they are declared (Item 13).

What's wrong with the following class?:

```
template<class T>
class Array {
public:
  Array(int low, int high);
  ...
private:
  vector<T> data;
  size_t size;
  int lBound, hBound;
};

template<class T>
Array<T>::Array(int low, int high)
:size(high - low + 1), lBound(low), hBound(high), data(size)
{}
```

Constructors, Destructors, and Assignment Operators

The problem in the previous code is that the constructor for `data` will be passed an uninitialized value.

The reason is that data members are initialized in the order they were *declared*, not in the order they are listed in the constructor.

This is to keep initialization consistent with destruction: class member destructors are always called in the inverse order of declaration.

Thus, to avoid confusion, you should always list members in an initialization list in the same order in which they are declared.

Constructors, Destructors, and Assignment Operators

Make sure base classes have virtual destructors (Item 14).

What's wrong with the following code?:

```
class EnemyTarget {
public:
  EnemyTarget() { }
  ~EnemyTarget() { }
};

class EnemyTank :public EnemyTarget {
public:
  EnemyTank() { ++numTanks; }
  ~EnemyTank() { --numTanks; }
  static size_t numberOfTanks() { return numTanks; }
private:
  static size_t numTanks;
};
size_t EnemyTank::numTanks = 0;

EnemyTarget *targetPtr = new EnemyTank;
delete targetPtr;
```

Constructors, Destructors, and Assignment Operators

The behavior of the previous code is undefined.

If you try to delete a derived class object through a base class pointer and the base class has a *nonvirtual* destructor, the result is undefined.

In practice, what will likely happen is that the derived destructor will not be called. As a result the value of `numTanks` will be incorrect.

Thus, if you plan to allow other classes to inherit from your class, its destructor should be virtual.

Constructors, Destructors, and Assignment Operators

Have operator= return a reference to *this (Item 15).

The purpose of allowing overloaded operators is to have user-defined classes mimic built-in types as closely as possible.

With built-in types, you can chain assignments together as follows:

```
int w, x, y, z;
w = x = y = z = 0;
```

To be consistent with this behavior, a class `C` should declare `operator=` as follows:

```
C& C::operator=(const C& rhs)
{
  ...
  return *this;
}
```

Any other way of writing `operator=` will be inconsistent with expected C++ behavior.

Constructors, Destructors, and Assignment Operators

Assign to all data members in operator= (Item 16).

If you don't write `operator=` for a class, it is generated automatically: each member of the class is simply copied.

If you *do* write `operator=`, C++ doesn't do anything for you. So you have to make sure and copy every data member. It is especially important to remember to update `operator=` if you add members to a class later.

If you have a derived class, you have to explicitly call the base class `operator=` (the same applies to the copy constructor):

```
Derived& Derived::operator=(const Derived& rhs)
{
  if (this == &rhs) return *this;
  Base::operator=(rhs);
  data = rhs.data;
  return *this;
}

Derived::Derived(const Derived& rhs)
  : Base(rhs), data(rhs.data) {}
```

Constructors, Destructors, and Assignment Operators

Check for assignment to self in operator= (Item 17).

What's wrong with this code?

```
class String {
public:
  String(const char *value;
  ~String();
  ...
  String& operator=(const String& rhs);
private:
  char *data;
}

String& String::operator=(const String& rhs)
{
  delete [] data;
  data = new char[strlen(rhs.data) + 1];
  strcpy(data, rhs.data);
  return *this;
}
```

Constructors, Destructors, and Assignment Operators

Suppose we have a self-assignment:

```
String a;
a = a;
```

Then, `a`'s data will get deleted and then used before it is reallocated.

It may seem silly to do a self-assignment, but it is not always obvious or easy to prevent.

The standard solution is to check for self-assignment in `operator=`:

```
C& C::operator=(const C& rhs)
{
  if (this == &rhs) return *this;
  ...
  return *this;
}
```

Classes and Functions: Design and Declaration

Declaring a new class in a program creaes a new type: class design is *type* design. To design effective types, you have to understand the issues involved:

- *How should objects be created and destroyed?*
  Constructors and Destructors.
- *How does object initialization differ from object assignment?*
  Constructors vs. `operator=`.
- *What does it mean to pass objects of the new type by value?*
  Copy constructor.
- *What are the constraints on legal values for the new type?* Error checking.
- *Does the new type fit into an inheritance graph?*
  What functions to declare `virtual`.
- *What kind of type conversions are allowed?*
  Implicit vs explicit type conversions.
- *What operators and functions make sense for the new type?*
  What to declare in the class interface.
- *What standard operators and functions should be explicitly disallowed?*
  Declare them private.
- *Who should have access to the members of the new type?*
  Public vs protected vs private, friends.
- *How general is the new type?* Should it be a template?

Classes and Functions: Design and Declaration

Strive for class interfaces that are complete and minimal (Item 18).

The *client interface* for a class is the interface that is accessible to the programmers who use the class. Typically these are class functions that are declared `public`.

A good interface is challenging. You are faced with the conflicting goals of keeping it simple and providing the functionality that clients want.

A good rule of thumb to balance these conflicting goals is to aim for a class interface that is *complete and minimal*.

A *complete* interface is one that allows clients to do anything they might reasonably want to do.

Why complete?

Classes and Functions: Design and Declaration

Strive for class interfaces that are complete and minimal (Item 18).

The *client interface* for a class is the interface that is accessible to the programmers who use the class. Typically these are class functions that are declared `public`.

A good interface is challenging. You are faced with the conflicting goals of keeping it simple and providing the functionality that clients want.

A good rule of thumb to balance these conflicting goals is to aim for a class interface that is *complete and minimal*.

A *complete* interface is one that allows clients to do anything they might reasonably want to do.

Why complete?

- Programmers expect the obvious functionality to be provided.
- But don't try to guess non-obvious future functionality.

Classes and Functions: Design and Declaration

A *minimal* interface is one with as few functions as possible, one in which no two member functions have overlapping functionality.

Why minimal?

Classes and Functions: Design and Declaration

A *minimal* interface is one with as few functions as possible, one in which no two member functions have overlapping functionality.

Why minimal?

- Too many functions makes a class difficult to understand.
- Functions with overlapping functionality can confuse the user.
- More difficult to maintain more functions.
- Long class definitions lead to long header files which increases build-time.

Classes and Functions: Design and Declaration

A *minimal* interface is one with as few functions as possible, one in which no two member functions have overlapping functionality.

Why minimal?

- Too many functions makes a class difficult to understand.
- Functions with overlapping functionality can confuse the user.
- More difficult to maintain more functions.
- Long class definitions lead to long header files which increases build-time.

Of course, there are times when you may want to make exceptions, but complete and minimal is a good guideline.

Classes and Functions: Design and Declaration

Differentiate among member functions, non-member functions, and friend functions (Item 19).

Here are some guidelines to help decide how to declare your functions.

1. *Virtual functions must be members*. If you have a function whose operation depends on where it is in the inheritance hierarchy, it has to be a virtual member function.
2. `operator<<` *and* `operator>>` *are never members*. These operators require a stream as their first argument and thus cannot be member functions. If they need to access private data, they will need to be made friend functions.
3. *Only non-member functions get type conversions on their left-most argument*. If a function requires type conversion on its left-most argument, make it a non-member function. If, in addition, it needs access to non-public members, make it a friend.
4. *Everything else should be a member function*.

To explain item 3, above, in more detail we consider an example.

Classes and Functions: Design and Declaration

Consider a class for rational numbers:

```
class Rational {
public:
  Rational(int numerator = 0, int denominator = 1);
  int numerator() const { return num; }
  int denominator() const { return den; }
  const Rational operator*(const Rational& rhs) const;
  ...
private:
  int num;
  int den;
};
```

What will happen when the following code is compiled?

```
Rational oneHalf(1, 2);
Rational result = oneHalf * 2;
result = 2 * oneHalf;
```

Classes and Functions: Design and Declaration

Consider a class for rational numbers:

```
class Rational {
public:
  Rational(int numerator = 0, int denominator = 1);
  int numerator() const { return num; }
  int denominator() const { return den; }
  const Rational operator*(const Rational& rhs) const;
  ...
private:
  int num;
  int den;
};
```

What will happen when the following code is compiled?

```
Rational oneHalf(1, 2);
Rational result = oneHalf * 2;
result = 2 * oneHalf;
```

The first use of `operator*` is fine, because `2` can be implicitly converted to `Rational` using the constructor. But the second will fail.

Classes and Functions: Design and Declaration

This inconsistent behavior is probably not what we want. We can fix it in two ways. The first fix is to make the constructor `explicit`:

```
class Rational {
public:
  explicit Rational(int numerator = 0,
                    int denominator = 1);
  ...
};
```

Now both uses of `operator*` will fail. `operator*` will only succeed if both arguments are true `Rational` objects. The other fix is to make `operator*` a non-member function:

```
const Rational operator*(const Rational& lhs,
                         const Rational& rhs)
{
  return Rational(lhs.numerator() * rhs.numerator(),
                  lhs.denominator() * rhs.denominator());
}
```

Now, both uses of `operator*` succeed. In addition, because `operator*` only uses the public interface of `Rational`, it does not need to be a friend.

Classes and Functions: Design and Declaration

Avoid data members in the public interface (Item 20).

There are some obvious reasons why this is a good idea.

- *Consistency*. If all access is via functions, clients don't have to remember whether to use function or data members.
- *Access control*. You can implement no access, read-only access, read-write access, even write-only access.
- *Functional abstraction*. You can change the class implementation without affecting clients.

Classes and Functions: Design and Declaration

Use const whenever possible (Item 21).

The `const` keyword allows you to enlist the compiler's aid in enforcing a constraint: namely that an object should not be modified.

The use of `const` can be confusing, so let's review how it is used. It's most basic use is to say that a variable of a basic type cannot be changed.

```
const x = 5;
x = 6; // error
```

It can get a little more confusing with pointers:

```
char* p = "ab";   // non-const pointer, non-const data
const char *p = "ab"; // non-const pointer, const data
char const *p = "ab"; // non-const pointer, const data
char * const p = "ab"; // const pointer, non-const data
const char * const p = "ab"; // const pointer,
                             // const data
```

The rule is: if `const` appears left of `*`, what's *pointed to* is constant; if `const` appears right of `*`, the *pointer itself* is constant.

Classes and Functions: Design and Declaration

Using `const` in function declarations can reduce errors without compromising safety or efficiency. For example, recall the declaration of `operator*` for the `Rational` class:

```
const Rational operator*(const Rational& lhs,
                         const Rational& rhs);
```

Why should the return type be `const`*?*

Classes and Functions: Design and Declaration

Using `const` in function declarations can reduce errors without compromising safety or efficiency. For example, recall the declaration of `operator*` for the `Rational` class:

```
const Rational operator*(const Rational& lhs,
                         const Rational& rhs);
```

Why should the return type be `const`*?*

Otherwise, you could write code like this:

```
Rational a, b, c;
(a * b) = c;
```

This would not be allowed by a built-in type. Making `operator*` return `const` disallows this for the `Rational` type too.

Classes and Functions: Design and Declaration

Finally, `const` member functions allow you to specify which member functions may be invoked on `const` objects. You can even overload based on whether a function is `const` or not:

```
class String {
public:
  ...
  char& operator[](int pos) { return data[pos]; }
  const char& operator[](int pos) const
  { return data[pos]; }
  ...
};

String s1 = "Hello";
cout << s1[0];

const String s2 = "Hello";
cout << s2[0];

s1[0] = 'x';  // fine
s2[0] = 'x';  // error
```

Classes and Functions: Design and Declaration

In C++, a `const` member function cannot change any of the object's data members. Occasionally, you may want to relax this restriction. The keyword `mutable` allows you to do this:

```
class DataSet
{
  ...
private
  mutable int average;
  mutable bool averageIsValid;
}
```

A `DataSet` object can be lazy about computing its average and the average can be computed on demand, even for `const` objects.