V22.0474-001 Software Engineering
Spring 2008

Lecture 15: Effective C++ II

Clark Barrett, New York University

Outline

Effective C++

- Classes and Functions: Design and Declaration
- Classes and Functions: Implementation

Sources for today's lecture:

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

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

What is wrong with the following code:

```
class Person {
public:
  Person(...);
  ~Person();
  ...
private:
  string name, address;
};

class Student :public Person {
public:
  Student(...);
  ~Student();
  Student returnStudent(Student s) { return s; }
private:
  string schoolName, schoolAddress;
}
```

Classes and Functions: Design and Declaration

Prefer pass-by-reference to pass-by-value (Item 22).

The main problem with the code (aside from not having a virtual destructor in the `Person` class) in the previous slide is that it uses pass-by-value. Consider the following simple use of the previous classes:

```
Student plato;
returnStudent(plato);
```

How many times are string constructors/destructors called during the call to `returnStudent`?

Classes and Functions: Design and Declaration

Prefer pass-by-reference to pass-by-value (Item 22).

The main problem with the code (aside from not having a virtual destructor in the `Person` class) in the previous slide is that it uses pass-by-value. Consider the following simple use of the previous classes:

```
Student plato;
returnStudent(plato);
```

How many times are string constructors/destructors called during the call to `returnStudent`?

The copy constructor for `Student` is called once for the parameter and once for the return value. Similarly for the destructor.

Each call to a constructor or destructor results in 4 calls to string constructors/destructors.

A total of 16 calls is made to string constructors/destructors.

Classes and Functions: Design and Declaration

An alternative implementation is:

```
const Student& returnStudent(const Student& s)
{ return s; }
```

With this implementation, there are no calls to any constructors or destructors.

Passing by reference also avoids the so-called *slicing problem*:

When a derived class object is passed by value as a base class object, all the derived object features are “sliced” off and you’re left with a base class object.

Passing by reference does have some complications:

- Aliasing
- Sometimes it’s wrong to pass by reference.
- It is more efficient to pass small objects (such as `int`s) by value.

Classes and Functions: Design and Declaration

What's wrong with this code?

```
const Rational& operator* (const Rational& lhs,
                           const Rational& rhs)
{
  Rational result(lhs.num * rhs.num, lhs.den * rhs.den);
  return result;
}
```

Classes and Functions: Design and Declaration

What's wrong with this code?

```
const Rational& operator* (const Rational& lhs,
                           const Rational& rhs)
{
  Rational result(lhs.num * rhs.num, lhs.den * rhs.den);
  return result;
}
```

The above code returns a reference to an object that no longer exists.

Classes and Functions: Design and Declaration

What's wrong with this code?

```
const Rational& operator* (const Rational& lhs,
                           const Rational& rhs)
{
  Rational result(lhs.num * rhs.num, lhs.den * rhs.den);
  return result;
}
```

The above code returns a reference to an object that no longer exists.

What about the following fix?

```
const Rational& operator* (const Rational& lhs,
                           const Rational& rhs)
{
  Rational *result =
    new Rational(lhs.num * rhs.num, lhs.den * rhs.den);
  return *result;
}
```

Classes and Functions: Design and Declaration

What's wrong with this code?

```
const Rational& operator* (const Rational& lhs,
                           const Rational& rhs)
{
  Rational result(lhs.num * rhs.num, lhs.den * rhs.den);
  return result;
}
```

The above code returns a reference to an object that no longer exists.

What about the following fix?

```
const Rational& operator* (const Rational& lhs,
                           const Rational& rhs)
{
  Rational *result =
    new Rational(lhs.num * rhs.num, lhs.den * rhs.den);
  return *result;
}
```

Who will delete this memory? This is a guaranteed memory leak!

Classes and Functions: Design and Declaration

OK, how about this?

```
const Rational& operator* (const Rational& lhs,
                           const Rational& rhs)
{
  static Rational result;
  result.num = lhs.num * rhs.num;
  result.den = lhs.den * rhs.den;
  return result;
}
```

Classes and Functions: Design and Declaration

OK, how about this?

```
const Rational& operator* (const Rational& lhs,
                           const Rational& rhs)
{
  static Rational result;
  result.num = lhs.num * rhs.num;
  result.den = lhs.den * rhs.den;
  return result;
}
```

This looks promising, but what about the following code?

```
Rational a, b, c, d;
if ((a * b) == (c * d))
...
```

Classes and Functions: Design and Declaration

OK, how about this?

```
const Rational& operator* (const Rational& lhs,
                           const Rational& rhs)
{
  static Rational result;
  result.num = lhs.num * rhs.num;
  result.den = lhs.den * rhs.den;
  return result;
}
```

This looks promising, but what about the following code?

```
Rational a, b, c, d;
if ((a * b) == (c * d))
...
```

The if-condition is always true, regardless of the values of `a`, `b`, `c`, or `d`!

Classes and Functions: Design and Declaration

Don’t try to return a reference when you must return an object (Item 23).

The right way to write this function is *not* to use pass-by-reference in the return value:

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

Classes and Functions: Design and Declaration

Choose carefully between function overloading and parameter defaulting (Item 24).

Suppose you want to allow a function to be called with different numbers of arguments.

How do you decide between using parameter defaulting and function overloading?

Classes and Functions: Design and Declaration

Choose carefully between function overloading and parameter defaulting (Item 24).

Suppose you want to allow a function to be called with different numbers of arguments.

How do you decide between using parameter defaulting and function overloading?

If

1. there is a reasonable default value, and
2. you always want to use the same algorithm,

then use default parameters. Otherwise, use function overloading.

Classes and Functions: Design and Declaration

What's wrong with this code?

```
void f(int x);
void f(string *ps);

f(NULL);
```

Classes and Functions: Design and Declaration

What's wrong with this code?

```
void f(int x);
void f(string *ps);

f(NULL);
```

Depending on how `NULL` is defined, this will either call the first function (if `NULL` is defined as `0`) or give a compile error (if `NULL` is defined as `((void*)0)`).

This is probably not what you want to happen, and there is no easy way to fix it, so

Avoid overloading on a pointer and a numerical type (Item 25).

Classes and Functions: Design and Declaration

What is the potential problem with the following code?

```
class B;

class A {
public:
  A(const B&);
};

class B {
public:
  operator A() const;
};
```

Classes and Functions: Design and Declaration

What is the potential problem with the following code?

```
class B;

class A {
public:
  A(const B&);
};

class B {
public:
  operator A() const;
};
```

The problem is *potential ambiguity*. Consider the following code:

```
void f(const A&);
B b;
f(b);
```

Because there are two ways to convert an object of type B to an object of type A, the compiler will complain.

Classes and Functions: Design and Declaration

Guard against potential ambiguity (Item 26).

There are other ways to generate ambiguity:

```
void f(int);
void f(char);
double d = 6.02;
f(d);
```

Multiple inheritance can easily result in ambiguity:

```
class Base1 {
  int doIt();
};

class Base2 {
  int doIt();
};

class Derived :public Base1, public Base2 { ...

Derived d;
d.doIt();
```

Classes and Functions: Design and Declaration

Explicitly disallow use of implicitly generated member functions you don't want (Item 27).

We talked about this a little bit last time. Suppose you want to create an array object but you don't want to allow assignment of one array to another:

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

Implicitly generated functions include default constructors, copy constructors, and assignment operators. Make sure you think about whether you want users of your class to have access to these functions.

Classes and Functions: Design and Declaration

Partition the global namespace (Item 28).

Namespaces allow you to safely combine your code with other peoples' code without worrying about name collisions.

We won't talk about it much in this class, partly because it shouldn't be an issue for your projects.

However, if you are working on a tool that you expect to provide as a library to clients, you should definitely use namespaces.

Classes and Functions: Implementation

Once you have a good interface, correctly implementing the member functions is usually straightforward.

However, there are a few pitfalls you need to watch for.

What is wrong with the following code?

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

String::operator char*() const
{ return data; }
```

Classes and Functions: Implementation

Avoid returning "handles" to internal data (Item 29).

The problem is that the `char*` operator function returns a *handle* to information that should be hidden.

As a result, a caller could modify the object, even if the object is `const`.

The same thing can happen with references:

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

Classes and Functions: Implementation

The general solution to these kinds of problems is either to make the function non-`const` or rewrite it so that no handle is returned (returning a `const` handle is often fine).

Even for non-`const` functions, returning handles is a bad idea because it violates abstraction and can lead to trouble, especially when temporary objects get involved.

Classes and Functions: Implementation

Postpone variable definitions as long as possible (Item 32).

In C++, every non-basic variable that is declared requires a call to a constructor and a destructor.

If a particular run of the program does not need the variable, this is wasted effort.

Consider the following example:

```
string encryptPassword(const string& password)
{
  string encrypted;
  if (!isValid(password)) {
    throw logic_error("Invalid Password");
  }
  encrypt(password, encrypted); // encrypt password
  return encrypted;
}
```

The string `encrypted` will not be used if there is an error. Therefore, it would be better to move the declaration of `encrypted` after the check for a valid password.

Classes and Functions: Implementation

Use inlining judiciously (Item 33).

When is inlining a good idea?

Classes and Functions: Implementation

Use inlining judiciously (Item 33).

When is inlining a good idea?

- Very simple code (no loops).
- Function is performance-critical.

Classes and Functions: Implementation

Use inlining judiciously (Item 33).

When is inlining a good idea?

- Very simple code (no loops).
- Function is performance-critical.

When is inlining a bad idea?

Classes and Functions: Implementation

Use inlining judiciously (Item 33).

When is inlining a good idea?

- Very simple code (no loops).
- Function is performance-critical.

When is inlining a bad idea?

All the rest of the time. Why?

- `virtual` functions cannot be inlined.
- inlining can increase code size.
- inlining can make debugging harder.
- inlining can increase compilation dependencies (see next item).

Classes and Functions: Implementation

Minimize compilation dependencies between files (Item 34).

If you've ever worked on a large project, you have probably had the experience of changing one line and having to wait for 10 minutes for everything to recompile.

C++ doesn't do a very good job of separating interfaces from implementations: often implementation details get put in header files and changing these causes a chain reaction in compilation.

This can be partly minimized by using good design patterns like the bridge pattern.

Another key principle is:

Make header files self-sufficient whenever it's practical and when it's not practical, make them dependent on class declarations, not class definitions.

We will look at a few applications of this principle.

Classes and Functions: Implementation

1. Avoid using objects when object references and pointers will do.

Consider the following definition:

```
#include "object.h"
class Wrapper {
private:
  Object x;
};
```

If you can reimplement this using a reference or pointer to Object, you don't have to include `object.h`:

```
class Object;
class Wrapper {
private:
  Object& x;
};
```

Classes and Functions: Implementation

2. Use class declarations instead of class definitions whenever you can.

In the last example, we were able to replace the class definition (found in the header file) with a class declaration because we changed the private data from an object to a reference to an object.

For member functions, we can do even better: you *never* need a class definition to declare a function using that class, even if the function passes or returns the class type by value:

```
class Date;
class DateManager {
public:
  Date getDate();
  void setDate(Date d);
...
};
```

Although you should question why `DateManager` isn't passing by reference instead of by value, the point is that this code compiles fine with just a simple declaration of `Date`.

Classes and Functions: Implementation

3. Don't include header files in your header files unless your headers won't compile without them.

This point is related to the previous two. Just because you reference class `A` in your header file, doesn't mean you need to include its definition.

Sometimes it can be tricky to figure out whether the definition is needed. Thus, the easy rule is that if it doesn't compile without the definition, include it, otherwise, don't.

This often means you will have to have more `#include` directives in your implementation files. That's fine. That's where they should be, not in your header files.