V22.0474-001 Software Engineering Spring 2008

Lecture 17: Effective C++ III

Clark Barrett, New York University

Outline

Effective C++

- Inheritance and Object-Oriented Design

Sources for today's lecture:

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

Meyers, Scott. *More Effective C++*.

Inheritance and Object-Oriented Design

`C++` provides a large assortment of *object-oriented building blocks*.

There is often *more than one way* to do the same thing.

Understanding *when* different features should be used can be challenging.

We will focus on understanding what the features of `C++` *really mean*.

Object-oriented design then becomes the process of:

1. *Understanding* what you want to say about your software system.
2. *Translating* that into the appropriate `C++` features.

Inheritance and Object-Oriented Design

Make sure public inheritance models "isa" (Item 35).

This is the most important rule in object-oriented programming with `C++`.

If you write that class `Derived` publicly inherits from class `Base`, you are saying:

- *Every object* of type `Derived` *is an* object of type `Base` (but not vice versa).
- `Base` represents a more general concept than `Derived`.
- `Derived` represents a more specialized concept than `Base`.
- *Anywhere* an object of type `Base` can be used, an object of type `Derived` can be used as well.
- Every *operation* that can be applied to an object of type `Base` can be applied to an object of type `Derived` as well.

Inheritance and Object-Oriented Design

Which are good candidates for public inheritance, and which are not?

- `StudentAddress` inherits from `ManhattanAddress`

  Probably a bad idea. Some students may not live in Manhattan, so it is not the case that every `StudentAddress` *is a* `ManhattanAddress`.

- `Student` inherits from `Person`

  Good. Not every person is a student, but every student is a person.

- `Penguin` inherits from `Bird`

  It depends. If `Bird` has a method called `fly`, then you have a problem. To fix the problem, split `Bird` into `FlyingBird` and `NonFlyingBird` classes.

- `Square` inherits from `Rectangle`

  Probably a bad idea. Even though every `Square` *is a* `Rectangle`, there are operations that can be done to a `Rectangle` but not to a `Square`, like `makeWider` or `makeTaller`.

Inheritance and Object-Oriented Design

Differentiate between inheritance of interface and inheritance of implementation (Item 36).

Consider the following base class:

```
#include<string>
class Shape {
public:
  virtual void draw() const = 0;
  virtual void error(const std::string& msg);
  int objectID() const;
  ...
};
```

What is being said by the way each method is declared?

Inheritance and Object-Oriented Design

What does a class interface say about the way its functions are inherited?

- Member function *interfaces* are always inherited.
- A *pure virtual function* says that derived classes inherit the *interface only*
- A simple (not pure) virtual function says that derived classes inherit the *interface plus a default implementation*.
- A nonvirtual function says that derived classes inherit the *interface plus a mandatory implementation*. The implementation of this function is an *invariant over specialization*.

Inheritance and Object-Oriented Design

What is disturbing about this code?

```
class B {
public:
  void mf();
  ...
};

class D :public B {
public:
  void mf();
  ...
};

D x;
B *px1 = &x;
D *px2 = &x;
px1->mf();
px2->mf();
```

Inheritance and Object-Oriented Design

We expect a function invoked via a pointer to `x` to always do the same thing.

This expectation can be violated if an inherited nonvirtual function is redefined, so

Never redefine an inherited nonvirtual function (Item 37).

The need to redefine an inherited nonvirtual function is an indicator of a contradiction in your design.

Suppose as above, `D` redefines the nonvirtual function `mf` defined by `B`.

- If `mf` really is an invariant over specialization but `D` still needs to redefine `mf`, then it cannot be the case that every `D` *is a* `B`.
- If every `D` really *is a* `B`, but `D` needs to redefine `mf`, then it's just not true that `mf` represents an invariant over specialization. In that case, `mf` should be virtual.
- If every `D` really *is a* `B`, and `mf` really is an invariant over specialization, then `D` shouldn't have to redefine `mf`.

Inheritance and Object-Oriented Design

What is disturbing about this code?

```
enum ShapeColor { RED, GREEN, BLUE };

class Shape {
public:
  virtual void draw(ShapeColor color = RED) const = 0;
  ...
};

class Rectangle :public Shape {
public:
  virtual void draw(ShapeColor color = GREEN) const;
  ...
};

Shape *pr = new Rectangle;
pr->draw();
```


Inheritance and Object-Oriented Design

In the previous example, `pr` points to a `Rectangle` object, but when the `draw` method is invoked, the default value from the base class `Shape` is used instead of the default value from the `Rectangle` class.

The reason is that virtual functions are *dynamically bound*, while default parameter values are *statically bound*. The moral of the story is:

Never redefine an inherited default parameter value (Item 38).

Of course, if you declared both functions nonvirtual, then both the function definition and the parameter value would be statically bound and would match.

Why is this a bad idea?

You would be violating the previous item: never redefine an inherited nonvirtual function!

Inheritance and Object-Oriented Design

Avoid casts down the inheritance hierarchy (Item 39).

What's a cast, you say? Thanks for asking (See Item 2 in *More Effective C++*).

In `C`, you can force the compiler to interpret an expression to be of a particular type as follows:

```
(type) expression
```

For example, if you want to convert a pointer to `char` into a pointer to `int` for some reason, you could do the following:

```
intPointer = (int*) charPointer;
```

The first thing you should know about casts is that they are ugly and should be avoided if possible.

The next thing to know is that `C++` has a more sophisticated set of constructs for performing casts.

Inheritance and Object-Oriented Design

There are four kinds of C++ casts. They all have the following format. Instead of

`(type) expression`,

use

`castkind``_cast<type>(expression)`,

where *`castkind`* is one of `static`, `const`, `dynamic`, or `reinterpret`.

- `static_cast`

  This is an all-purpose cast that can be used most anywhere the old C-style cast could be used. For example,

```
int* intPointer;
char* charPointer = 'A';

intPointer = static_cast<int*>(charPointer);
```


Inheritance and Object-Oriented Design

- `const_cast`

  This cast can *only* be used to cast away the `const`ness of an object (or the `volatile`ness of an object).

  Consider the following example:

```
class Widget { ... };

void update(Widget *pw);

Widget w;
const Widget& cw = w;

update(&cw); // error - cannot pass a const Widget*
             // in place of a Widget*

update(const_cast<Widget*>(&cw)); //ok - constness cast away
```

  Note that this is the only cast that can cast away `const`ness. If you try to cast away `const`ness using a `static_cast`, you will get an error.

Inheritance and Object-Oriented Design

- `dynamic_cast`

  This is primarily used to perform *safe casts* down an inheritance hierarchy:

```
class SpecialWidget :public Widget { ... };

void display(SpecialWidget *psw);

Widget *pw = new SpecialWidget;
...
update(dynamic_cast<SpecialWidget*>(pw));
```

  The difference between using `dynamic_cast` and `static_cast` in this situation is that `dynamic_cast` actually checks to make sure the object being pointed to is of the target type. If not, it returns `NULL` (or throws an exception if you are casting a reference).

- `reinterpret_cast`

  This is used for non-portable implementation-dependent casts. You should avoid using it unless you know what you are doing. Even then, you should probably avoid using it.

Inheritance and Object-Oriented Design

Now that we've reviewed casting and shown you how to cast down the inheritance hierarchy, let me remind you of the item we are discussing:

Avoid casts down the inheritance hierarchy (Item 39).

Consider the following example:

```
#include<string>
class BankAccount {
public:
  BankAccount(const std::string& owner);
  virtual ~BankAccount();
  ...
};

class SavingsAccount :public BankAccount {
public:
  SavingsAccount(const std::string& owner);
  ~SavingsAccount();
  void creditInterest();
  ...
};
```

Inheritance and Object-Oriented Design

Now, suppose the bank keeps a list of all its accounts and you want to credit each account with interest.

What's wrong with this code?

```
#include<list>
using namespace std;
list<BankAccount*> allAccounts;
...
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end(); ++p) {
  (*p)->creditInterest();
}
```

The `creditInterest` method only belongs to the subclass `SavingsAccount`.

Inheritance and Object-Oriented Design

Now, suppose the bank keeps a list of all its accounts and you want to credit each account with interest.

What's wrong with this code?

```
#include<list>
using namespace std;
list<BankAccount*> allAccounts;
...
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end(); ++p) {
  (*p)->creditInterest();
}
```

The `creditInterest` method only belongs to the subclass `SavingsAccount`.

What about this fix?

```
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end(); ++p) {
  static_cast<SavingsAccount*>(*p)->creditInterest();
}
```


Inheritance and Object-Oriented Design

```
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end(); ++p) {
  static_cast<SavingsAccount*>(*p)->creditInterest();
}
```

The main problem with this code is that if someone now adds a new type of account, say a `CheckingAccount`, the behavior will be undefined.

The other problem is that you will be tempted to fix it like this:

```
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end(); ++p) {
  if (isSavingsAccount(*p))
    static_cast<SavingsAccount*>(*p)->creditInterest();
  else
    static_cast<CheckingAccount*>(*p)->creditInterest();
}
```

What's wrong with the new code?

Inheritance and Object-Oriented Design

Here's what Scott Meyers has to say about it:

> *Anytime you find yourself writing code of the form, "if the object is of type T1, then do something, but if it's of type T2, then do something else," slap yourself (EC++, p. 176).*

In `C++`, the preferred method for type-dependent behavior is virtual functions.

How can we solve our bank account problem using virtual functions?

- Add a new class `InterestBearingAccount` and have the bank maintain a list of those instead of a list of `BankAccount`s.
- Add a virtual method `creditInterest` to `BankAccount` with a default implementation that does nothing.

Inheritance and Object-Oriented Design

Fine, you say, but suppose I'm just a lowly programmer who is handed a list of `BankAccount` objects by a big bank company who refuses to change their class design. What then?

Well, in that case, you have to use casts. But at least use a *safe downcast*:

```
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end(); ++p) {
  if (SavingsAccount *psa = dynamic_cast<SavingsAccount*>(*p))
    psa->creditInterest();
  else if (CheckingAccount *pca =
           dynamic_cast<CheckingAccount*>(*p))
    pca->creditInterest();
  else
    error("Unknown account type");
}
```

Note the use of just-in-time variable declaration and the check for an unknown account type. It's not as pretty as using virtual functions, but it works.

Inheritance and Object-Oriented Design

Suppose you want to create a `Set` class that is slightly different from the one in the standard library.

You decide to implement `Set` using the `list` class from the standard library. Your implementation might look like this:

```
#include <list>
template<class T>
class Set :public std::list<T> {
public:
  bool member(const T& item) const;
  void insert(const T& item) const;
  ...
};
```

What's wrong with this design?

This design says that a `Set` *is a* `list`. But a `list` may contain duplicates, whereas a `Set` may not.

Because it is not the case that a `Set` *is a* `list`, public inheritance is the *wrong* way to model the relationship.

Inheritance and Object-Oriented Design

What you really want to do is to *implement* `Set` *in terms of* `list`. This brings us to the next item:

Model "has-a" or "is-implemented-in-terms-of" through layering (Item 40).

Layering is the process of building one class on top of another class by having the layering class contain an object of the layered class as a data member. For example,

```
#include<string>
class Address { ... };
class PhoneNumber { ... };
class Person {
public:
  ...
private:
  std::string name;
  Address address;
  PhoneNumber voiceNumber;
  PhoneNumber faxNumber;
};
```


Inheritance and Object-Oriented Design

The `Person` class is said to be *layered on top of* the `string`, `Address`, and `PhoneNumber` classes.

The `Person` class demonstrates the *has a* relationship. Fortunately, most people do not confuse *is a* with *has a*.

On the other hand, our `Set` example demonstrates an *is implemented in terms of* relationship. Here's the right way to do it.

```
#include <list>
template<class T>
class Set {
public:
  bool member(const T& item) const;
  void insert(const T& item) const;
  ...
private:
  std::list<T> rep;
};
```

It's worth mentioning that layering creates compile-time dependencies. *What could you do to eliminate these?*

Inheritance and Object-Oriented Design

Suppose you want to create classes representing stacks of objects. You'll need several different classes, one for stacks of `int`s, one for stacks of `string`s, etc.

Suppose you also want to design classes representing many different kinds of cats. You'll need multiple classes because each breed of cat is a little bit different.

These two problems sound similar. The question is whether they result in similar designs.

How would you solve these two design problems?

The first should be done using *templates*, the second using *inheritance*.

Differentiate between inheritance and templates (Item 41).

- A template should be used to generate a collection of classes when the type of the objects *does not* affect the behavior of the class's functions.
- Inheritance should be used for a collection of classes when the type of the objects *does* affect the behavior of the class's functions.

Inheritance and Object-Oriented Design

Use private inheritance judiciously (Item 42).

Private inheritance behaves differently from public inheritance:

- Compilers will *not* convert a derived class object into a base class object if the inheritance relationship between the classes is private.
- All members inherited from a private base class become private members of the derived class.

So, what does it mean?

> *Private inheritance means is-implemented-in-terms-of*.

Since layering also means is-implemented-in-terms-of, how do you choose between them?

> *Choose layering whenever you can; use private inheritance whenever you must.*

As an example, if the class you want to use has protected methods, the only way you can use those protected methods is by inheriting from the class. If you need to use protected methods but the relationship isn't *is a*, use private inheritance.

Inheritance and Object-Oriented Design

Use multiple inheritance judiciously (Item 43).

Multiple inheritance leads to a host of complexities. One of the most basic is ambiguity. Consider the following:

```
class Lottery {
public:
  virtual int draw();
};
class GraphicalObject {
public:
  virtual int draw();
};
class LotterySimulation: public Lottery,
                         public GraphicalObject {
  ...
};

LotterySimulation *pls = new LotterySimulation;
pls->draw();                  // error - ambiguous
pls->Lottery::draw();         // ok
pls->GraphicalObject::draw(); // ok
```


Inheritance and Object-Oriented Design

The problem in the previous slide is ambiguity. Other problems caused by multiple inheritance include:

- How do you decide whether to make a base class virtual or not?
- Passing constructor arguments to virtual base classes.
- Dominance of virtual functions.

You should beware of using multiple inheritance unless you understand all of these issues.

However, there are some cases where multiple inheritance can be useful. One example is if you want to inherit an interface publicly and an implementation privately (and layering is not an option).

Still, if you have a choice, it is often better to redesign the inheritance hierarchy than to rely on multiple inheritance.

Inheritance and Object-Oriented Design

Say what you mean; understand what you're saying (Item 44).

This is a summary of how C++ constructs map to design-level ideas:

- Public inheritance means *isa*.
- Private inheritance means *is-implemented-in-terms-of*.
- Layering means *has-a* or *is-implemented-in-terms-of*.

For public inheritance, we have the additional mappings:

- A pure virtual function means that only the function's interface is inherited.
- A simple virtual function means that the function's interface plus a default implementations is inherited.
- A nonvirtual function menas that the function's interface plus a mandatory implementation is inherited.