Review

Last week

• Classes
• Encapsulation and Inheritance
• Initialization and Finalization
• Dynamic Method Binding
• Abstract Classes
• Simulating First-Class Functions
Outline

Effective C++

- Constructors, Destructors, and Assignment Operators
- Classes and Functions: Design and Declaration
- Classes and Functions: Implementation
- Inheritance and Object-Oriented Design

Sources for today’s lecture:


Barrett. Lecture notes, Fall 2008.
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?*
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';
    }
}
Consider the following code:

```java
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:

```cpp
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:

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

Which is the better constructor definition?:

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

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

```cpp
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.
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:

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

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

```cpp
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):

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

```cpp
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;
}
Suppose we have a self-assignment:

```cpp
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=`:

```cpp
C& C::operator=(const C& rhs)
{
    if (this == &rhs) return *this;
    ...
    return *this;
}
```
Declaring a new class in a program creates 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?
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?*

- 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;

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

```cpp
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:

```cpp
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.
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.
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.

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

It can get a little more confusing with pointers:

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

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

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

Otherwise, you could write code like this:

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

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

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

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

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

```cpp
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.
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;
}
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:

```c++
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.
An alternative implementation is:

```cpp
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 `ints`) by value.
What's wrong with this code?

```cpp
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?

```cpp
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?

```cpp
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?_

```cpp
class Rational {
private:
    int num;
    int den;

public:
    Rational() : num(0), den(1) { }
    Rational(int n, int d) : num(n), den(d) { }
    ~Rational() { }
    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;
    }
    // Other member functions...
};
```

This looks promising, but what about the following code?

```cpp
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!
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:

```cpp
const Rational operator* (const Rational& lhs,
                          const Rational& rhs)
{
    return Rational(lhs.n * rhs.n, lhs.d * rhs.d);
}
```
Classes and Functions: Design andDeclaration

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.
What's wrong with this code?

```c
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).**
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:

```cpp
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.
Guard against potential ambiguity (Item 26).

There are other ways to generate ambiguity:

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

Multiple inheritance can easily result in ambiguity:

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

class Base2 {
    int doIt();
};

class Derived : public Base1, public Base2 {
    Derived d;
d. doIt();
};
```
Explicitly disallow use of implicitly generated member functions you don’t want (Item 27).

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

```cpp
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.
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; }
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:

```cpp
class String {
public:
    ...  
    char& operator[](int index) const
        { return data[index]; } 
private:
    char *data;
};
```
The general solution to these kinds of problems is either to make the function non-
\texttt{const} or rewrite it so that no handle is returned (returning a \texttt{const} handle is often fine).

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

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

Consider the following definition:

```cpp
#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`:

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

```cpp
#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*. 

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?*

```cpp
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:

\[(\text{type}) \text{ expression}\]

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

\[\text{intPointer} = (\text{int*}) \text{ charPointer};\]

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

\[(\text{type}) \ \text{expression},\]

use

\[\text{castkind\_cast}\text{<type>}(\text{expression}),\]

where \text{castkind} is one of \text{static}, \text{const}, \text{dynamic}, or \text{reinterpret}.

- \text{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);
**const_cast**

This cast can only be used to cast away the constness of an object (or the volatileness of an object).

Consider the following example:

```cpp
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 constness. If you try to cast away constness 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:

```cpp
#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?*

```cpp
#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?*

```cpp
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 BankAccounts.

- 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:

```cpp
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.
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:

```cpp
#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,

```cpp
#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.

```cpp
#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?*
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 `strings`, 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.
Use multiple inheritance judiciously (Item 43).

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

```cpp
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 implementation is inherited.
- A nonvirtual function means that the function’s interface plus a mandatory implementation is inherited.