CPSC 427a: Object-Oriented Programming

Michael J. Fischer

Lecture 18
November 9, 2010

STL and Polymorphism

Derivation from STL containers

Common wisdom on the internet says not to inherit from STL containers.

For example,
http://en.wikipedia.org/wiki/Standard_Template_Library says,

> *"STL containers are not intended to be used as base classes (their destructors are deliberately non-virtual); deriving from a container is a common mistake."*

This reflects Rule 35 of Sutter and Alexandrescu,

> *"Avoid inheriting from classes that were not designed to be base classes."*

Replacing authority with understanding

C++ is a complicated and powerful language.

Some constructs such as classes are used for several different purposes.

What is appropriate in one context may not be in another.

Simple rules will not make you a good C++ programmer. Thought, understanding, and experience will.

Two kinds of derivation

C++ supports two distinct kinds of derivation:

- Simple derivation.
- Polymorphic derivation.

```
class A { ... };
class B : public A { ... };
```

We say `B` is **derived** from `A`, and `B` **inherits** members from `A`.

Each `B` object has an `A` object embedded within it.

The derivation is **simple** if no members of `A` are `virtual`; otherwise it is `polymorphic`.

How are they the same?

With both kinds of derivation, a function of the base class `A` can be **overridden** by a function in `B`.

In both cases, one can create and delete objects of class `B`.

Both `A`'s and `B`'s destructor are called when a `B` object is deleted.

```
#include <iostream>
class A { public:
  ~A() { std::cout << "A's destructor called" << std::endl; }
};
class B: public A { public:
  ~B() { std::cout << "B's destructor called" << std::endl; }
};
int main() { B bobj; }
```

Output:
```
B's destructor called
A's destructor called
```

What is simple derivation good for?

Some uses for simple derivation.

- Code sharing. A common base can be extended in different directions through derivation.
- Creating a new API to system resources (e.g., `12-StopWatch` demo).
- Increasing modularity through layering.

With simple derivation, the derived class is the public interface.

Often `protected` or `private` derivation is used to hide the base class from the users of the derived class.

What are the problems with simple derivation?

- Several objects derived from the same base type have little in common except for the embedded base type object in each.
- A base type pointer can only access the embedded base type object. The rest of the derived object is present but invisible. This is called **slicing**, where the derived part is conceptually "sliced off".

What is polymorphic derivation good for?

- Polymorphic derivation allows for variability among objects with a common interface.
- The base class (possibly pure abstract) defines the interface.
- Each derived class defines a variant or implementation of the interface.

Some uses for polymorphic derivation.

- Heterogeneous containers. Example: An array of different kinds of employees.
- A mechanism whereby old code can call new code. By deriving from a predefined interface, existing functions that call virtual functions of the base class end up invoking new user-provided code.

What are the problems of polymorphic derivation?

Every polymorphic base class (containing even one virtual function) adds a runtime **type tag** to each instance.

This costs in both time and space.

- Time: Each call to a virtual function goes through a run-time dispatch table (the **vtable**).
- Space: Each instance of a polymorphic object contains a type tag, which takes extra space.
- Every polymorphic base class should have a `virtual` destructor.

Contrasts between simple and polymorphic derivation

Simple derivation:

- Low cost.
- Extends the base class.
- Derived class is the public interface; base class is a helper.
- Slicing is generally avoided as being not useful.

Polymorphic derivation:

- Higher cost.
- Implements the base class (in possibly multiple ways).
- Base class is the public interface; derived classes are helpers.
- Slicing is encouraged; virtual functions provide access to underlying derived class objects.

Containment as an alternative to simple derivation

Often the same class can be implemented using either containment or derivation.

Derivation:

```
class A { ... f() ... };
class B: public A { ... g() { f() ... } };
```

A's public member functions are inherited by B.

Containment:

```
class A { ... f() ... };
class B { private: A a; ... g() { a.f() ... }
   public: f() { return a.f(); } };
```

Access to A's public member functions requires a "pass-through" function for delegation.

Argument for containment

Containment is a more distant relationship than derivation.

Less coupling between classes is safer and less error-prone.

Using containment, derived class is explicit about what is exported.

For more info, see http://www.gotw.ca/publications/mill06.htm.

STL container as a base class

We apply these concepts to STL base classes.

Base classes are simple, not polymorphic (no virtual functions, no virtual destructor).

This means that they should only be used with simple derivation. They are not suitable as base classes for polymorphic derivation.

Often containment is preferable, but the large number of member functions they support makes it cumbersome to get the same degree of functionality in the derived class as comes "for free" with derivation.

Can I turn an STL container into a polymorphic base class?

Yes, sort of. Here's the idea.

```
#include <iostream>
#include <vector>
using namespace std;
class MyVectorInt : public vector<int> {
public:
  MyVectorInt() : vector<int>() {}
   virtual ~MyVectorInt() {
      cout << "Base class destructor is called" << endl; }
};
class Derived : public MyVectorInt {
public:
   ~Derived() {
      cout << "Derived destructor is called" << endl; }
};
```

A polymorphic base class

`MyVectorInt` is a polymorphic base class with virtual destructor and can be used as such.

```
int main() {
  MyVectorInt* p;  // a polymorphic pointer
  Derived* obj = new Derived();  // a derived object
  p = obj;          // ok to assign
  delete p;         // ok to delete; destructors called
}
```

Dynamic cast

It is always okay to cast a pointer to a derived class into a pointer to the base class, as in the previous example.

The reverse is only semantically meaningful if the allocated type of the object actually is the type to which it is being cast. In that case, one can use a `dynamic_cast` to effect the converstion.

```
MyVectorInt* p;
Derived* q;
...
q = dynamic_cast<Derived*>(p);
```

`dynamic_cast` returns `NULL` if `p` is pointing to something that does not have dynamic type `Derived*`.

Exceptions

Exceptions

An *exception* is an event that prevents normal continuation.

Exceptions may be due to program errors or data errors, but they may also be due to external events:

- File not found.
- Insufficient permissions.
- Network failure.
- Read error.
- Out of memory error.

How to respond to different kinds of exceptions is application-dependent.

Exception handling

When an exception occurs, a program has several options:

- Try again.
- Try something else.
- Give up.

Problem: Exceptions are often detected at a low level of the code. Knowledge of how to respond resides at a higher level.

C-style solution using status returns

The C library functions generally report exceptions by returning status values or error codes.

Advantages: How to handle exception is delegated to the caller.

Disadvantages:

- Every caller must handle every possible exception.
- Exception-handling code becomes intermingled with the "normal" operation code, making program much more difficult to comprehend.

C++ exception mechanism

C++ exception mechanism is a means for a low-level routine to report an exception directly to a higher-level routine.

This separates exception-handling code from normal processing code.

An exception is reported using the keyword `throw`.

An exception is handled in a `catch` block.

Each routine in the chain between the reporter and the handler is exited cleanly, with all destructors called as expected.