CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 22
April 19, 2016

Code Reuse

Reusable code

One of the major goals of C++ is code reusability.

The desire to reuse code occurs in two very different scenarios.

Sharing Different parts of a single application needs the same or similar code blocks. The code should be written once and shared by the parts that need it. Mechanisms for code sharing include functions and (non-polymorphic) derivation.

Libraries A code base is made available for others to use in their applications, e.g., the C++ Standard Library. Useful mechanisms include polymorphic derivation and templates.

Problems with reusing code

The problem with code reuse is that one rarely wants to reuse the exact same piece of code. Rather, one wants similar code but specialized to a particular application.

For example, most useful functions take parameters which tell the function what data to compute on. Different applications can call the function with their own data.

Similarly, containers such as `vector` make sense for many different kinds of objects. The STL container `vector<T>` allows the generic vector to be specialized to any suitable type `T` object.

How to allow variability

Code reusability becomes the problem of bridging the gap between abstract general code and specific concrete application.

- For functions, the various function call mechanisms bridge the gap.
- With polymorphic derivation, a base class pointer pointing to a specific derived class object bridges the gap.
- With templates, the template parameter bridges the gap.

In all three cases, application-specific data must satisfy the constraints required by the general code.

Specifying constraints

One of the important mechanisms in C++ for specifying contraints is the type system.

- For functions, the types of the actual arguments must be compatible with the declared types of the parameters. Violations of these constraints can be detected at compile time.
- With polymorphic derivation, the type system also allows for some constraint checking, but a dynamic downcast (convert from pointer-to-base to pointer-to-derived) requires runtime checking.
- With templates, specifying and checking the constraints is more difficult. We explore some of the ways this can be done.

Linear Containers

Demo 19-Virtual

Linked list code was introduced in demo 09-BarGraph, where a Row was a linked list of Item*, where an Item contained exam information for a particular student.

Demo 19-Virtual extracted the linked list code from Row and called it Linear. The specific Item class from 09-BarGraph was renamed Exam, and the type Item was reserved for the kind of object that could be put in a linked list.

Sharing in `19-Virtual`

`19-Virtual` observes that stacks and queues are very similar data structures.

- They are both linear lists of items.
- Both support operations `put()` and `pop()` that allow items to be inserted into and removed from the list.
- The only difference is where in the list new items are inserted.

The common code is in the base class `Linear`. The derived classes `Stack` and `Queue` override virtual base class functions as needed for their specializations.

Abstract containers

Both `Stack` and `Queue` are examples of list containers that support four operations: put, pop, peek, and print.

Class `Container` is an abstract base class with a virtual desctructor and four pure abstract functions put, pop, peek, and print.

`Linear` is derived from `Container`. This ensures that any generic code for dealing with containers can handle `Linear` objects.

However, `Container` is general on only one dimension. It is still specific to containers of `Item*` objects.

Ordered Containers

Demo 22a-Multiple

The purpose of demo `22a-Multiple` is to generalize the linear containers of demo `19-Virtual` to support lists of items that are sorted according to a data-specific ordering.

It does this by adding `class Ordered` and `Item`, creating two ordered containers of type `class List` and `class PQueue`, and extending the code appropriately.

`Ordered` base class

`Ordered` is an abstract class (interface) that promises items can be ordered based on an associated key.

It promises functions:

- A function `key()` that returns the key associated with an item.
- Comparison operators `<` and `==` that compare the derived item `*this` with an argument key.

Use:

```
class Item : public Exam, Ordered { ... };
```

Note: We can use private derivation because every function in `Ordered` is abstract and therefore must be overridden in `Item`.

class Item

`Item` is publicly derived from `Exam`, so it has access to `Exam`'s public and protected members.

It fulfills the promises of `Ordered` by defining:

```
bool
operator==(const KeyType& k) const { return key() == k; }
bool
operator< (const KeyType& k) const { return key() < k; }
bool
operator< (const Item& s)    const { return key() < s.key(); }
```

`KeyType` is defined with a `typedef` in `exam.hpp` to be `int`.

Container base class

We saw the Container abstract class in demo 19-Virtual. It promises four functions:

```
virtual void     put(Item*)       =0; // Put in Item
virtual Item*    pop()            =0; // Remove Item
virtual Item*    peek()           =0; // Look at Item
virtual ostream& print(ostream&) =0; // Print all Items
```

Use: `class Linear : Container { ... };`

Additions to `Linear`

The meaning of `put()`, `pop()`, and `peek()` for ordered lists is different from the unordered version, even though the interface is the same.

The concept of a **cursor** is introduced into `Linear` along with new virtual functions `insert()` and `focus()` for manipulating the cursor.

`peek()` and `pop()` always refer to the position of the cursor. `put()` inserts into the middle of the list in order to keep the list properly sorted.

class Linear

`Linear` implements general lists through the use of a *cursor*, a pair of private `Cell` pointers `here` and `prior`.

Protected `insert()` inserts at the cursor.
Protected `focus()` is virtual and must be overridden in each derived class to set the cursor appropriately for insertion.

Cursors are accessed and manipulated through protected functions `reset()`, `end()`, and `operator ++()`.

Use:
`List::insert(Cell* cp) {reset(); Linear::insert(cp);}`
inserts at the beginning of the list.

class PQueue

PQueue inserts into a sorted list.

```
void insert( Cell* cp ) {
    for (reset(); !end(); ++*this) {  // find  insertion spot.
        if ( !(*this < cp) )break;
    }
    Linear::insert( cp );             // do the insertion.
}
```

Note the use of the comparison between a PQueue and a Cell*.

This is defined in linear.hpp using the cursor:

```
bool operator< (Cell* cp) {
  return (*cp->data < *here->data); }
```

Multiple Inheritance

What is multiple inheritance

Multiple inheritance simply means deriving a class from two or more base classes.

Recall from demo `22a-Multiple`:

```
class Item : public Exam, Ordered { ... };
```

Here, `Item` is derived from both `Exam` and from `Ordered`.

Object structure

Suppose class `A` is multiply derived from both `B` and `C`.
We write this as `class A : B, C { ... };`.

Each instance of `A` has "embedded" within it an instance of `B` and an instance of `C`.

All data members of both `B` and `C` are present in the instance, even if they are not visible from within `A`.

Derivation from each base class can be separately controlled with privacy keywords, e.g.:
`class A : public B, protected C { ... };`.

Diamond pattern

One interesting case is the diamond pattern.

```
class D                 { ... x ... };
class B : public D      { ... };
class C : public D      { ... };
class A : public B, C   { ... };
```

Each instance of `A` contains *two* instances of `D`—one in `B` and one in `C`.

These can be distinguished using qualified names.
Suppose `x` is a public data member of `D`.
Within `A`, can write `B::D::x` to refer to the first copy, and `C::D::x` to refer to the second copy.

Template Example

Using templates with polymorphic derivation

To illustrate templates, I converted `22a-Multiple` to use template classes. The result is in `22b-Multiple-template`.

There is much to be learned from this example.
Today I point out only a few features.

Container class hierarchy

As before, we have `PQueue` derived from `Linear` derived from `Container`.

Now, each of these have become template classes with parameter `class T`.
`T` is the item type; the queue stores elements of type `T*`.

The main program creates a priority queue using
`PQueue<Item> P;`

Item class hierarchy

As before, we have `Item` derived from `Exam, Ordered`.

`Item` is an *adaptor* class.
It bridges the requirements of `PQueue<T>` to the `Exam` class.

Ordered template class

`Ordered<KeyType>` describes an abstract interface for a total ordering on elements of abstract type `KeyType`.

`Item` derives from `Ordered<KeyType>`, where `KeyType` is defined in `exam.hpp` using a `typedef`.

An `Ordered<KeyType>` requires the following:

```
virtual const KeyType& key() const                          =0;
virtual bool            operator <  (const KeyType&) const =0;
virtual bool            operator == (const KeyType&) const =0;
```

That is, there is the notion of a sort key. `key()` returns the key from an object satisfying the interface, and two keys can be compared using `<` and `==`.

Alternative `Ordered` interfaces

As a still more abstract alternative, one could require only comparison operators on abstract elements (of type `Ordered`). That is, the interface would have only two promises:]

```
virtual bool operator <  (const Ordered&) const  =0;
virtual bool operator == (const Ordered&) const  =0;
```

This has the advantage of not requiring an explicit key, but it's also less general since keys are often used to locate elements (as is done in the demo).