CPSC 427a: Object-Oriented Programming

Michael J. Fischer

Lecture 20
November 10, 2011

Ordered Container

Demo `20a-Multiple`

The purpose of demo `20a-Multiple` is to generalize the linear containers of demo `18c-Virtual` to support ordered lists of items.

It does this by adding `class Ordered`, 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`.

Container base class

We saw the Container abstract class in demo 18c-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 { ... };
```

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.

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 `20a-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.

Handling Circularly Dependent Classes

Tightly coupled classes

Class B *depends on* class A if B refers to elements declared within class A or to A itself.

The class B definition must be read by the compiler after reading A.

This is often ensured by putting `#include "A.hpp"` at the top of file `B.hpp`.

A pair of classes A and B are *tightly coupled* if each depends on the other.

It is not possible to have both read after the other.
Whichever the compiler reads first will cause the compiler to complain about undefined symbols from the other class.

Example: List and Cell

Suppose we want to extend a cell to have a pointer to a sublist.

```
class Cell {
  int data;
  List* sublist;
  Cell* next;
  ...
};
class List {
  Cell* head;
  ...
};
```

This won't compile, because `List` is used (in `class Cell`) before it is defined. But putting the two class definitions in the opposite order also doesn't work since then `Cell` would be used (in `class List`) before it is defined.

Circularity with #include

Circularity is less apparent when definitions are in separate files.

File list.hpp:

```
#pragma once
#include "cell.hpp"
class List { ... };
```

File cell.hpp:

```
#pragma once
#include "list.hpp"
class Cell { ... };
```

File main.cpp:

```
#include "list.hpp"
#include "cell.hpp"
int main() { ... }
```

What happens?

In this example, it appears that `class List` will get read before `class Cell` since `main.cpp` includes `list.hpp` before `cell.hpp`.

Actually, the opposite occurs. The compiler starts reading `list.hpp` but then jumps to `cell.hpp` when it sees the `#include "cell.hpp"` line.

It jumps again to `list.hpp` when it sees the `#include "list.hpp"` line in `cell.hpp`, but this is the second attempt to load `list.hpp`, so it only gets as far as `#pragma once`. It then resumes reading `cell.hpp` and processes `class Cell`.

When done with `cell.hpp`, it resumes reading `list.hpp` and processes `class List`.

Resolving circular dependencies

Several tricks can be used to allow tightly coupled classes to compile. Assume `A.hpp` is to be read first.

1. Suppose the only reference to `B` in `A` is to declare a pointer. Then it works to put a "forward" declaration of `B` at the top of `A.hpp`, for example:
   ```
   class B;
   class A { B* bp; ... };
   ```
2. If a function defined in `A` references symbols of `B`, then the *definition* of the function must be moved outside the class and placed where it will be read after `B` has been read in, e.g., in the `A.cpp` file.
3. If the function needs to be inline, this is still possible, but it's much trickier getting the inline function definition in the right place.

Template Example

Using templates with polymorphic derivation

To illustrate templates, I converted `20a-Multiple` to use template classes. The result is in `20b-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).