CPSC 427a: Object-Oriented Programming

Michael J. Fischer

Lecture 16
November 2, 2010

Multiple Inheritance

What is multiple inheritance

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

Example:

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

See demo `16-Multiple`

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

16-Multiple-template

To illustrate templates, I converted 16-Multiple to use template classes.

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->Linear->Container`.
Now, each of these are template classes with parameter `<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->Exam, Ordered`.

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

Ordered template class

`Ordered` describes an abstract interface for an abstract key type.
It becomes a template class with type parameter `KeyType`.

`Item` derives from `Ordered<int>`.

Alternative `Ordered` interfaces

The code presents two alternative interfaces:

1. The primary interface requires every client to define `<` and `==` with elements of type `KeyType`.
2. The alternative interface requires only comparison operators on abstract elements (of type `Ordered`). However, to define those operators one must also have available some function for obtaining the data to be used in the comparison – hence the `key()` function.

A real application would choose one interface or the other and go with it.
Both are in the code for comparison.

The C++ Standard Library

A bit of history

C++ standardization.

- C++ standardization began in 1989.
- ISO and ANSI standards were issued in 1998, nearly a decade later.
- The standard covers both the C++ language and the standard library (everything in namespace `std`).
- The standardization process continues as the language evolves and new features are added.

The standard library was derived from several different sources.

STL (Standard Template Library) portion of the C++ standard was derived from an earlier STL produced by Silicon Graphics (SGI).

Containers

A container stores a collection of objects of arbitrary type `T`.

The basic containers in STL are:

- `vector` – a dynamic array
- `deque` – a double-ended queue
- `list` – a doubly linked list
- `map` – an associative array of key/value pairs with unique keys
- `set` – a sorted collection of unique values
- `multimap` – an associative array of key/value pairs with duplicate keys allowed
- `multiset` – a sorted collection of values with multiplicity

Common container operations

All containers share a large number of operations.

Operations include creating an empty container, inserting, deleting, and copying elements, scanning through the container, and so forth.

Liberal use is made of operator definitions to make containers behave as much like other C++ objects as possible.

Containers implement value semantics, meaning type `T` objects are copied freely within the containers.

If copying is a problem, store pointers instead.

vector<T>

A `vector<T>` is a growable array of elements of type `T`.

You must `#include <vector>`.

Elements can be accessed using standard subscript notion.

Inserting at the beginning or middle of a `vector` takes time $O(n)$.

Example:

```
vector<int> tbl(10);  // creates length 10 vector of int
tbl[5] = 7;           // stores 7 in slot #5
cout << tbl[5];       // prints 7
tbl[10] = 4;          // illegal, but not checked!!!
cout << tbl.at(5);    // prints 7
tbl.at(10) = 4;       // illegal and throws an exception
tbl.push_back(4);     // creates tbl[10] and stores 4
cout << tbl.at(10);   // prints 4
```