CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 10
October 3, 2016

Bar Graph Demo (continued)

Introduction to the
C++ Standarad Library

Handling Circularly Dependent Classes

Bar Graph Demo (continued)

graph.cpp

Points to note:

- The `for`-loop in the constructor does not properly handle error conditions and can get into an infinite loop.
  You should test yourself to be sure you know how to fix this problem.
- The constructor has an allocation loop. The destructor has a corresponding deallocation loop.
- `bar[index]->insert( initials, score );`
  shows the use of a subscript and a pointer dereferencing in the same statement.
- Why do we need the `*` in
  `out << *bar[k] <<"\n";`

Design issues for `Graph` class

1. Note the use of the C preprocessor to allow preprocessor macro `NESTED` to cause compilation in two different ways.
2. Could we declare `bar` as `Row& bar[BARS]`? How might this affect the program?
3. Should `initials` be a string?
4. Why is there a potential infinite loop? What should be done about it?

row.hpp

Points to note:

- This file contains two *tightly coupled* classes, `Cell` and `Row`.
- The line `friend class Row` in `Cell` gives `Row` permission to access private data and methods of `Cell`.
- A class can give friendship. It cannot take friendship.
- The `Cell` constructor combines two operations that could be separated:
  1. It creates a new `Item` from a C-string and an integer;
  2. It creates a new fully initialized `Cell` containing as `data` a pointer to the newly-created `Item`.
- A `Row` has a `head` that points to the first `Cell` in a linked list.

`row.cpp`

Points to note:

- There is some clever coding in the `Row` constructor.
  Is this a good design?
- The destructor in `Row` deletes the entire linked list of `Cell`s.
  Why shouldn't this be done in the `Cell` destructor?
- `insert` creates a new `Cell` and puts it on the linked list.
  Where does it go?
- In `Row::print()`, the code reaches through `Cell` into `Item::print()`.
  This violates the rule, *"Don't talk to strangers."*
  - Is it okay in this context?
  - Why or why not?
  - What would the alternative be? [Hint: Delegation.]

rowNest.hpp

This is an alternative definition of class `Row` with the same public interface and behavior but different internal structure.

Points to note:

- In `row.hpp`, `Cell` is a top-level class in which everything is private. The `friend` declaration allows `Row` to use it.
- In `rowNest.hpp`, `Cell` is declared as a private class inside of `Row`, but everything in `Row` is public. Since only `Row` can access the class name, nobody else can access it.
- In all other respects, `row.hpp` and `rowNest.hpp` are identical.
- To determine which is used, change the `#include` in `graph.hpp`.

Discussion of `row.hpp` vs. `rowNest.hpp`

What are the questions you should be asking yourself when deciding which version you prefer?

item.hpp

This is a data class. In C, one would use a `struct`, but C++ permits tighter semantic control.

Points to note:

- The fields are private. They are initialized by the constructor and never changed after that.
- The only use made of those fields is by `print()`. Hence there is no need even for getter functions.
- `Item` could have been defined as a subclass of class `Row`. What are the pros and cons of such a decision?

Introduction to the C++ Standarad 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).

Some useful classes

Here are some useful classes that you have already seen:

- `string` – a character string designed to act as much as possible like the primitive data types such as `int` and `double`.
- `iostream`, `ifstream`, `ofstream` — buffered reading and writing of character streams.
- `istringstream` – permits input from an in-memory string-like object.
- `vector<T>` – creates a growable array of objects of type `T`, where `T` can be any type.

Class `stringstream`

A `stringstream` object (in the default case) acts like an `ostream` object.

It can be used just like you would use `cout`.

The characters go into an internal buffer rather than to a file or device.

The buffer can be retrieved as a `string` using the `str()` member function.

`stringstream` example

Example: Creating a label from an integer.

```
#include <sstream>
...
int examScore=94;
stringstream ss;
string label;
ss << "Score=" << examScore;
label = ss.str();
cout << label << endl;
```

This prints `Score=94`.

vector

`vector<T> myvec` is something like the C array `T myvec[]`.

The element type `T` can be any primitive, object, or pointer type.

One big difference is that a `vector` starts empty (in the default case) and it grows as elements are appended to the end.

Useful functions:

- `myvec.push_back( item )` appends `item` to the end.
- `myvec.size()` returns the number of objects in `myvec`
- `myvec[k]` returns the object in myvec with index `k` (assuming it exists.) Indices run from 0 to size()-1.

Other operations on `vectors`

Other operations include creating an empty vector, inserting, deleting, and copying elements, scanning through the vector, and so forth.

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

Vectors implement value semantics, meaning type `T` objects are copied freely within the vectors.

If copying is a problem, store pointers instead.

`vector` examples

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
```

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 each 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.