CPSC 427a: Object-Oriented Programming

Michael J. Fischer

Lecture 8
September 28, 2010

Bar Graph Demo

We look at the Bar Graph demo from Chapter 8 of the textbook.

graph.hpp

```
class Graph {
  private:
    Row* bar[BARS]; // List of bars (aggregation)
    void insert( char* name, int score );
  public:
    Graph ( istream& infile );
    ~Graph();
    ostream& print ( ostream& out );
    // Static functions are called without a class instance
    static void instructions() {
      cout  << "Put input files in same directory "
               "as the executable code.\n";
    }
};
inline ostream& operator<<( ostream& out, Graph& G) {
    return G.print( out );
}
```

Notes: `graph.hpp`

- A `Graph` consists of an array of *pointers* to `bar`s.
- We say that it *aggregates* the `bar`s because they are associated with the `Graph` but are not contained within it.
- The `bar`s must be allocated when the `Graph` is created and deallocated when the `Graph` is destroyed. This is done with constructors and destructors.
- The only constructor builds a `Graph` by reading an open `istream`.
- The method `insert` is used by the constructor. Hence it is declared `private`. It computes which bar an exam score belongs to and then puts it there.
- `instructions` is a static method. It is called using `Graph::instructions()`.

```
Graph::Graph( istream& infile ) {
  char initials[4];
  int score;

  // Create bars
  for (int k=0; k<BARS; ++k) bar[k] = new Row(k);

  // Fill bars from input stream
  for (;;) {
    infile >> ws;  // Skip leading whitespace before get.
    infile.get(initials, 4, ' ');  // Safe read.
    if (infile.eof()) break;
    infile >> score;          // No need for ws before >> num.
    insert (initials, score); // *** POTENTIAL INFINITE LOOP
  }
}
```

Notes: graph.cpp

This implements four functions.

- `Graph()` first creates 11 `bar`s and links them to the spine `bar[]`. This forms a 2D array.
- `Graph()` next reads the scores and fills the graph.
- `ws` skips over leading whitespace.
- `get(initials, 4, ' ')` is a safe way to read initials.
- The destructor `~Graph()` deletes the 11 `bar`s.
- `insert()` divides the scores 0...99 into 10 intervals.
- `print()` delegates the printing of each bar to `Row::print()`.

Private class for use by Row.
Note friend declaration and private constructor.

```
class Cell
{
  friend class Row;
  private:
    Item* data;  // Pointer to one data Item (Aggregation)
    Cell* next;  // Pointer to next cell in row (Association)

    Cell (char* d, int s, Cell* nx) {
      data = new Item(d, s);
      next = nx;
    }
    ~Cell (){ delete data; cerr <<"  Deleting Cell " <<"\n"; }
};
```

row.hpp

Public class represents one bar of the bar graph

```
class Row { // Interface class for one bar of the bar graph.
  private:
    char label[10];  // Row header label
    Cell* head;      // Pointer to first cell of row
  public:
    Row ( int n );
    ~Row ();
    void insert ( char* name, int score );  // delegation
    ostream& print ( ostream& os );
};
```

Notes: `row.hpp`

A `Row` is a list of `Item`. It is implemented by a linked list of `Cell`.

- The `Cell` class is private to `Row`. Nothing but its name is visible from the outside.
- `friend class Row` allows `Row` functions to access the private parts of `Cell`.
- Since all constructors of `Cell` are private, any attempt to allocate a `Row` from outside will fail.
- Each `Cell` is initialized when it is created.
- `Row::head` points to the first cell of the linked list.

Notes: row.cpp

- Row k is labeled by the length 9 string "`k0..k9:␣␣`". E.g., $k = 4 \Rightarrow$ label is "`40..49:␣␣`".
- Label is produced by a safe copy and modify trick:

```
strcpy( label, " 0.. 9:  " );
label[0] = label[4] = '0'+ rowNum;
```

- `'0'+rowNum` converts an integer in [0..9] to the corresponding ASCII digit.
- Assignment in C++ returns the L-value of its left operand. In C, it returns the R-value of its right operand.
- `Cell` created and inserted into linked list in one line!

Nested classes: rowNest.hpp

Alternative to Row.

Puts entire Cell class definition inside of class Row.

Now Cell is private in Row, but everything inside of class Cell is public.

This obviates the need for Cell to grant friendship toRow and also completely hides Cell—even the name is hidden.

Interface is same, so can substitute

```
#include "rowNest.hpp"
```

for

```
#include "row.hpp"
```

in graph.hpp and everything still works!

Storage management

Storage classes

C++ supports three different storage classes.

1. `auto` objects are created by variable and parameter declarations. (This is the default.)
2. `static` objects are created and initialized at load time and exist until program termination.
3. `new` creates *dynamic* objects. The exist until explicitly destroyed by `delete` or the program terminates.

Assignment and copying

The assignment operator `=` is implicitly defined for all types.

- `b=a` does a shallow copy from `a` to `b`.
- Shallow copy on objects means to copy all data members from one object to the other.
- Call-by-value uses assignment to copy actual argument to function parameter.
- If object contains pointer data members, the pointer is copied but *not* the object it points to. This results in *aliasing*—multiple pointers to the same object.

Static data members

A static class variable must be *declared* and *defined*.

- A static class member is declared by preceding the member declaration by the qualifier `static`.
- A static class member is defined by having it appear in global context with an initializer but *without* `static`.
- Must be defined *only once*.

Example

In `mypack.hpp` file, inside class definition:

```
class MyPack {
  static int instances; // count # instantiations
```

In `mypack.cpp` file:

```
int MyPack::instances = 0;
```

Static function members

Function members can also be declared `static`.

- As with static variables, the are declared inside class by prefixing `static`.
- They may be defined either inside the class (as inline functions) or outside the class.
- If defined outside the class, the `::` prefix must be used and the word `static` omitted.

Five common kinds of failures

1. **Memory leak**—Dynamic storage that is no longer accessible but has not been deallocated.
2. **Amnesia**—Storage values that mysteriously disapper.
3. **Bus error**—Program crashes because of an attempt to access non-existant memory.
4. **Segmentation fault**—Program crashes because of an attempt to access memory not allocated to your process.
5. **Waiting for eternity**—Program is in a permanent wait state or an infinite loop.

Read the textbook for examples of how these happen and what to do about them.

Bells and whistles

Optional parameters

The same name can be used to name several different member functions if the *signatures* (types and/or number of parameters) are diffent. This is called overloading.

Optional parameters are a shorthand way to declare overloading.

Example

```
int myfun( double x, int n=1 ) { ...  }
```

This declaresdefines two methods:

```
int myfun( double x ) {...}
int myfun( double x, int n ) {...}
```

The body of the definition of both is the same.
If called with one argument, the second parameter is set to 1.

const

`const` declares a variable (L-value) to be readonly.

```
const int x;
int y;
const int* p;
int* q;

p = &x;  // okay
p = &y;  // okay
q = &x;  // not okay -- discards const
q = &y;  // okay
```

const implicit argument

`const` should be used for member functions that do not change data members.

```
class MyPack {
private:
  int count;
public:
  // a get function
  int getCount() const { return count; }
...
};
```

Operator extensions

Operators are shorthand for functions.

Example: `<=` refers to the function `operator <=()`.

Operators can be overloaded just like functions.

```
class MyObj {
  int count;
  ...
  bool operator <=( MyObj& other ) const {
      return count <= other.count; }
};
```

Now can write `if (a <= b) ...` where `a` and `b` are of type `MyObj`.