CPSC 427a: Object-Oriented Programming

Michael J. Fischer

Lecture 7
September 22, 2011

IO Demos

Introduction to Classes

BarGraph Demo

- Specification
- `graph.hpp`
- `graph.cpp`
- `row.hpp`
- `row.cpp`
- `rowNest.hpp`

IO Demos

Handling data errors and end of file

Section 3.7 of the textbook contains a demo program that illustrates how to handle data errors and end of file using C++ I/O. It has three parts that illustrate

- How to use `get()` to read text lines from a file.
- How to use `getline()` to do the same thing.
- How to read numbers from a file.

See 06-IOdemo.

How to write a test program

The 06-IOdemo was written in C-style using three global functions: `use_get()`, `use_getline()`, and `use_nums()`.

I rewrote the demo

- to eliminate the use of underscores in multipart names;
- to illustrate the use of C++ classes as lexical containers for gathering and isolating related code.

Here, each test is encapsulated within its own class.

The only responsibility of `main()` is to process the command line arguments and initiate the tests.

See 07-IOdemo-new.

Introduction to Classes

(Textbook, Chapter 4)

Classes, visibility, functions, inline

We covered much of the material from sections 4.1 and 4.2 in lectures 2 and 3.

The textbook covers it in greater depth, so be sure to also read the book.

Bar Graph Demo

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

Bar graph sample input and output

Input:

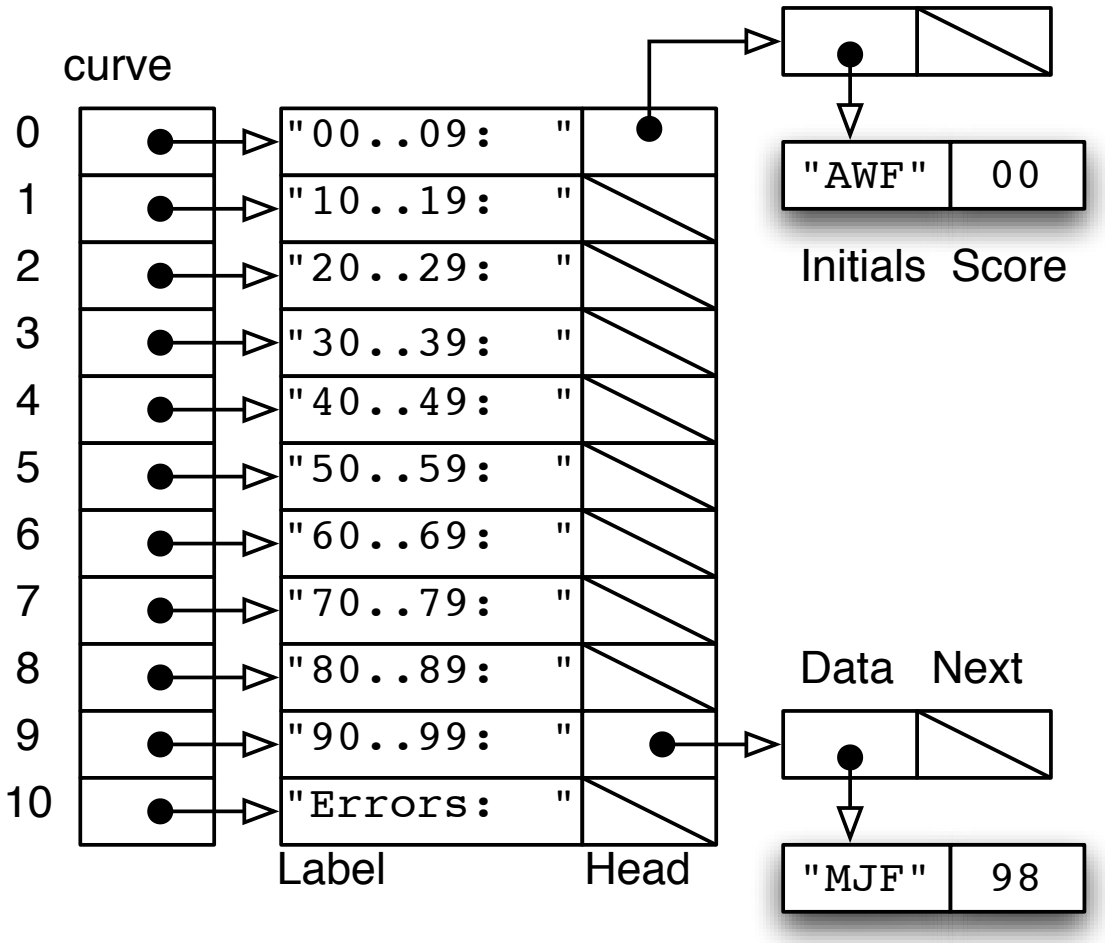
```
AWF  00
MJF  98
FDR  75
RBW  69
GBS  92
PLK  37
ABA  56
PDB  71
JBK  -1
GLD  89
PRD  68
HST  79
ABC  82
AEF  89
ALA  105
```

Output:

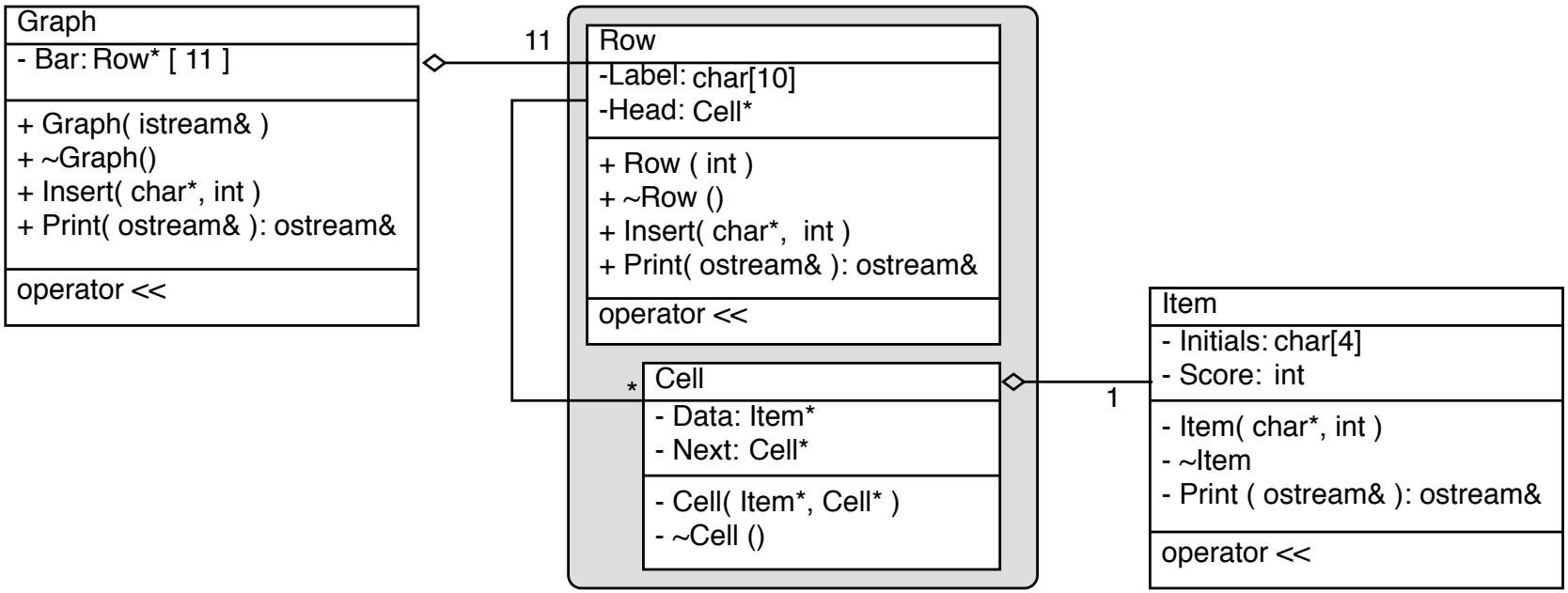
```
Name of data file: bar.in
File is open and ready to read.

00..09:  AWF 0
10..19:
20..29:
30..39:  PLK 37
40..49:
50..59:  ABA 56
60..69:  PRD 68 RBW 69
70..79:  HST 79 PDB 71 FDR 75
80..89:  AEF 89 ABC 82 GLD 89
90..99:  GBS 92 MJF 98
Errors:  ALA 105 JBK -1
```

Bar graph data structure



UML Diagram


Graph
- Bar: Row* [11]
+ Graph(istream&)
+ ~Graph()
+ Insert(char*, int)
+ Print(ostream&): ostream&
operator <<
11
Row
-Label: char[10]
-Head: Cell*
+ Row (int)
+ ~Row ()
+ Insert(char*, int)
+ Print(ostream&): ostream&
operator <<
*
Cell
- Data: Item*
- Next: Cell*
- Cell(Item*, Cell*)
- ~Cell ()
1
Item
- Initials: char[4]
- Score: int
- Item(char*, int)
- ~Item
- Print (ostream&): ostream&
operator <<

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.cpp

```
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()`.

row.hpp

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!