Object-Oriented Principles and Practice / C++

Alice E. Fischer

May 4, 2013

C++ gives us several ways to initialize class data members.

- A data member may be initialized in the class declaration.
- There are special rules for initialization of static class members because they must be initialized at load time.
- A member may be initialized by a ctor prior to execution of a constructor.
- A member may be initialized by a constructor.

C++11 introduced the third and fourth options.

Initialization in the Class Declaration

```
class Tester {
private:
    // primitive types ========================
    int n1 = -1;
    static const int n3 = 17;

    // class types ============================
    Item it1, it2;
    Item it3 = {.99, 1};
    const Item it4 = {2, 2};
```

C++11 gives us new ways to initialize arrays.

```
class Tester {
private:
    static int a0[3];
    static constexpr int a1[3]{1,2,3};
    int a2[3] = {8};
    int a3[3];
    int* a4;
    int* a5 = new int[3]{5,6,7};
public:
    Tester():  it2{.5, 2}, a3(), a4(new int[3]{7,8,9}) {}
}
const int Tester::a1[3];
int Tester::a0[3]{4,5,6};
```

These things do not compile.

A const is something that is constant at block-entry time.
A constexpr is an expression that is constant at compile time.

- Error: non-const static data member must be initialized out of line.

```
static int n2 = 99;
```

- Error: constexpr variable 'it5' must be initialized by a constant expression

```
static constexpr Item it5 = {1.01, 10};
```

- Error: in-class initializer for static data member of type 'const int [3]' requires 'constexpr' specifier.

```
static const int a1[3]1,2,3;
```

Disasters

Chapter 6 and Chapter 3.7
Five Kinds of Failure
Causes and Cures
End of File and Error Handling

Five Kinds of Failure

- Memory management error: failure to allocate or deallocate at the right time.
- Amnesia: it WAS stored there, but now it is gone.
- Bus error: attempt to access a machine location that does not exist.
- Segmentation error: attempt to access a machine location that belongs to someone else.
- Waiting for eternity: no progress is happening.

Causes and Cures

- Shallow copy / deep delete.
- Dangling pointers.
- Storing things in limbo.
- The wrong way to delete.
- Walking on memory.
- Parentheses are not square brackets.
- NULL pointers
- Stream input errors.

In-class Examples.

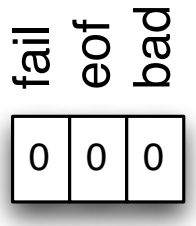
- Assign
- Dangle
- Limbo
- Delete
- Walk
- Brackets

Stream State Flags

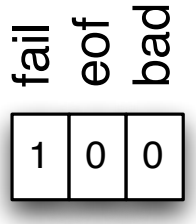
See Chapter 3.7.

Three flags are part of every stream object:

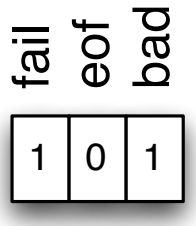
Normal, good read.
rdstate() returns 0
good() returns true



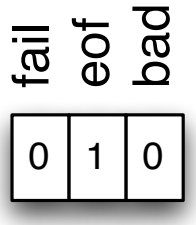
Conversion error occurred.
rdstate() returns 4
good() returns false



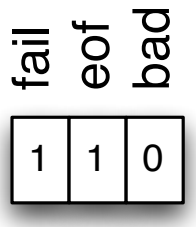
Hardware error occurred.
rdstate() returns 5
good() returns false



End of file occurred
but data was read.
rdstate() returns 2
good() returns false



End of file occurred
no data was read.
rdstate() returns 2
good() returns false



Using the Stream Bits

To handle all possible situations, make tests in the right order.

```
for(;;) {
  instr >> number;
  if (instr.good()) cout <<number <<endl; // process item.
  else if (instr.eof() ) break;
  else if (instr.fail()) {   // Without these three lines,
     instr.clear();          // an alphabetic input char
     instr.ignore(1);        // causes an infinite loop.
  }
  else if (instr.bad())      // Abort if unrecoverable.
    fatal( "Bad error while reading input stream." );
}
```

Bar Graph: an Example

Chapter 8.3
Specification and Data Structure
The main program
Class Item
Class Graph

Model and Viewer

A client typically provides a good description of the input for a project and a rough description of the desired functionality.

- A designer's specification gives an unambiguous description of the input, the desired functionality, and the output.
- We can, and often do, display the output in more than one way: those ways are called views of the data.
- Underneath all the views is a model of the data. A model is the set of variables that stores and organizes all the information necessary to create one or more views.
- A skilled program designer selects a model that will store the information efficiently, will perform acceptably well at run time, and will be easy for the coder to use.

Associate–Aggregate–Compose

Suppose you have two classes, A and B, and that each A object contains two B objects.

- If the two B's are part of the memory area allocated for A (no pointers here) then A *composes* B. The B object is born and dies with the A object.
- If the two B's can exist before A is created and A's constructor initializes A to points to them, then A *aggregates* B. The three parts were not born at the same time, but A is not a complete object unless it points at two B's.
- If the B's can be created either before or after A, and A can be complete whether it points to zero, one or two of them, then A *associates* with B.

The Bar Graph App

A teacher I know likes to see the student exam scores spread out on a bar graph.

- She wants a program that will take a file of student initials and scores (0..100), and display them with one line per 10-point group.
- She provided the following I/O guidance for the developer.
- She says that the order of the scores on a given line does not matter.

The developer's job is to create a model that can store the data efficiently and an output display that the teacher will like.

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: exam1.txt

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

Average score (excluding errors):  69.6
```

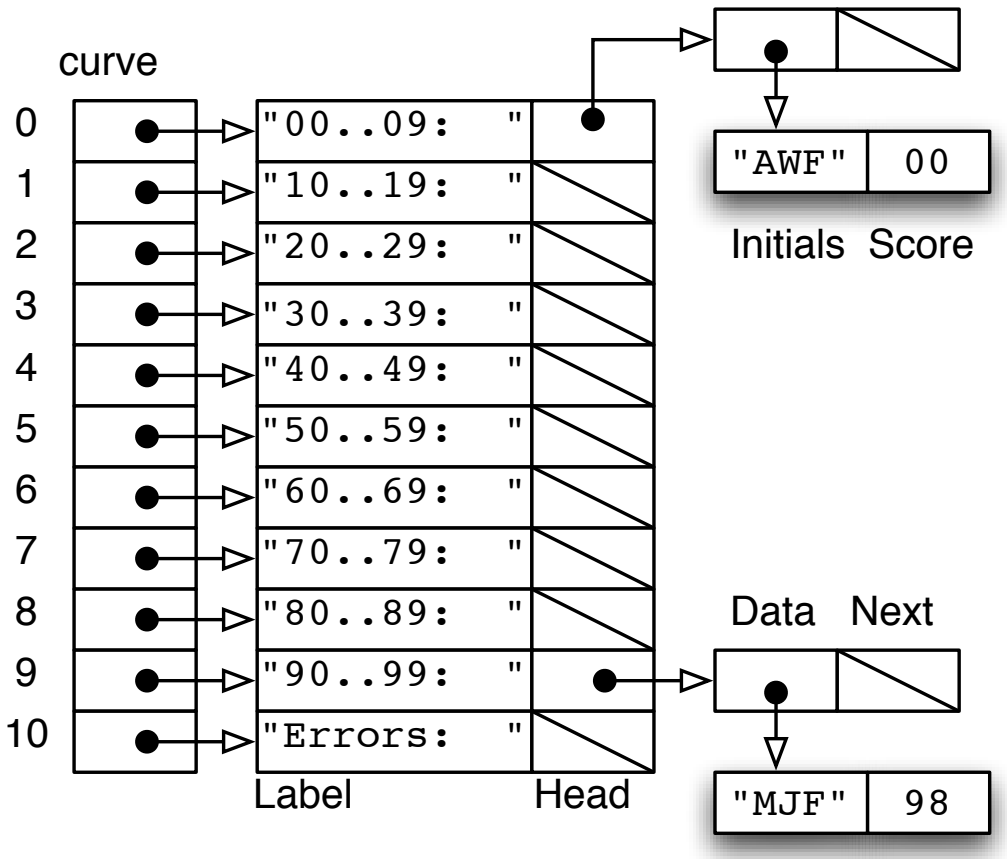
Modeling a Bar Graph

Clearly, this is more than the simple bar graphs that can be made using a spreadsheet. To model it we need:

- A class to store the information for one student (4 chars and one int).
- A class to represent one row of the display, with a label and a variable number of students on its list. The list of student scores could be a vector or a linked list. A linked list was chosen.
- A class to represent all the rows and the row for errors. Clearly, this should be an ordinary array because we know exactly how many rows are needed.

Bar graph data structure

We will build this data structure:

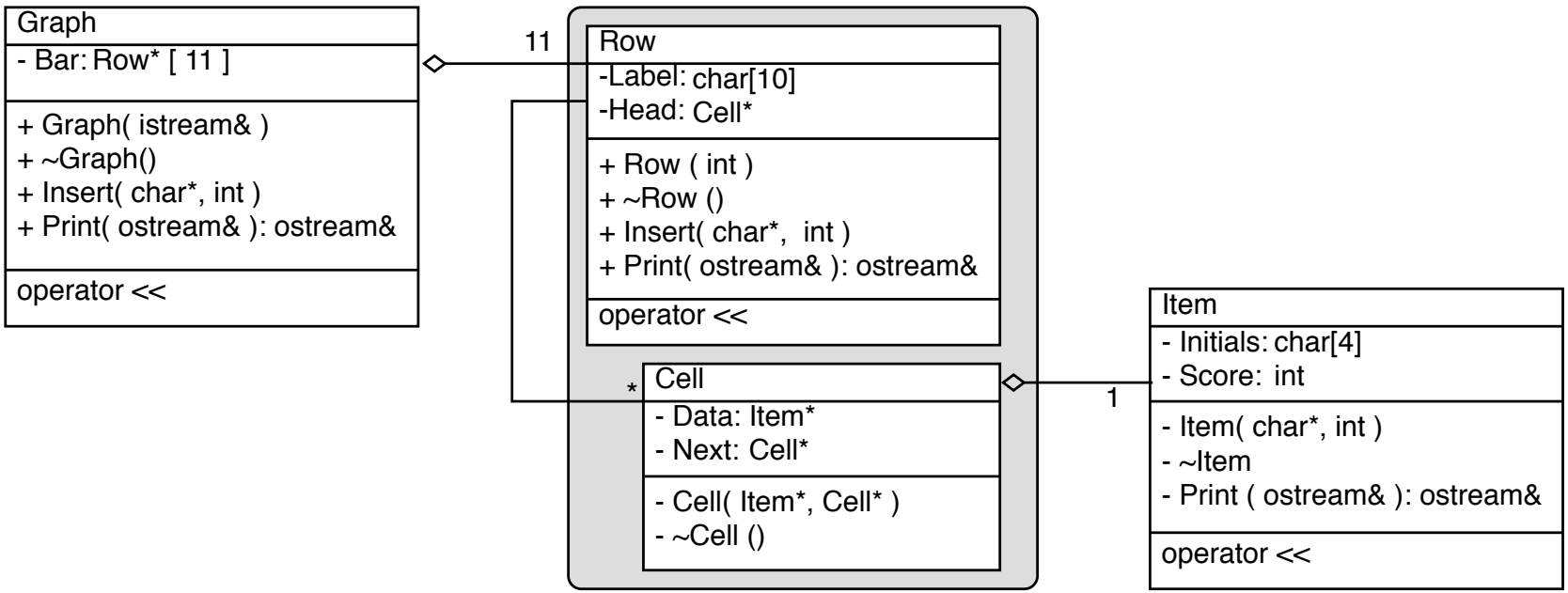


Notes on the Data Structure Diagram

A data structure diagram shows the relationships, at one moment of runtime, among the objects in the model.

- Our `Graph` consists of an array of *pointers* to `Rows`.
- Each row is a linked list of Cells. Each cell *composes* three initials and a score.
- We say that the Graph *aggregates* the `Rows` because they are associated with the `Graph` but are not contained within it.
- The `Rows` must be allocated and initialized when the `Graph` is created and deallocated when the `Graph` is destroyed. This is done with constructors and destructors.

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 <<

Notes on the UML Diagram

A UML diagram shows the static relationships among the classes in an application.

- Graph is the primary, or first, class for this application.
- Each class is represented by a rectangle with 3 (sometimes 4) layers: Name, data members, function members, and associated other things.
- Row, Cell, and Item are the other classes. Row and Cell, together, implement the linked list.
- The line connecting Graph to Row has a white diamond on one end and 11 on the other. This indicates that Graph *aggregates* 11 Rows.

More Notes on the UML Diagram

- The Row and Cell classes are enclosed in a gray box. This indicates that they are a tightly-coupled pair of classes, and that one of these classes *gives friendship* to the other.
- The asterisk on the line from Row to Cell tells you that Row is the master and Cell is the slave: Each Row has many Cells.
- The line with no diamond on the end represents *association*.
- The line connecting Cell to Item has a white diamond and a 1 on it. This means that each Cell *aggregates* one Item.
- Members marked by $+$ are public, those marked by $-$ are private.

Bar Graph: main program

Page 82 This program reads a file of exam data and prints a bar-graph of results. Each data line consists of a student's 3 initials and a score between 0 and 100.

- The main program interacts with the user to print instructions, get the file name, and opens the input stream.
- It then creates a Graph object (line 23) and prints the resulting graph (line 25).
- All of the action occurs within the Graph class and other classes.
- Main programs should be kept short, like this, or even shorter.
- Note the call on the static class function on line 14.

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 is defined here as an array of 11 bar pointers, where each bar will store student scores in a single 10-point range. (Page 86)

- A `Graph` consists of an array of *pointers* to `bars`.
- We could use an array of Rows instead, and will look at this alternative next week.
- We say that it *aggregates* the `bars` because they are associated with the `Graph` but are not contained within it.
- The `bars` must be allocated when the `Graph` is created and deallocated when the `Graph` is destroyed. This is done with constructors and destructors. `istream`.

Graph class declaration, continued

Page 86

- The only constructor builds a `Graph` by reading an open
- The method `insert` is used by the constructor and *should not* be used anywhere else. Hence it is declared `private`. It computes which bar an exam score belongs to and then puts it there.
- A public static function is provided to print instructions. The intention is that this function should be called before calling the class constructor. The `instructions` function is called using `Graph::instructions()`.

Notes: 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 continued

Pages 86–87 The constructor builds the entire graph.

- This is a 12-line function, quite long for a constructor. An alternative design would be to move the file input loop into a separate, private getScores function and call it from the constructor.
- `Graph()` first creates 11 `bars` 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.

Graph class implementation, continued.

- Because the graph is declared as an array of row pointers, the Graph destructor contains a loop to delete 11 Row*s.
- If the class used Rows instead of Row*s, this delete loop would be unnecessary (and wrong).
- The insert function is private because it will be called from the constructor and should not be called at any other time.
- `insert()` divides the scores 0...99 into 10 intervals, to find which bar each score belongs in. It then calls the `insert()` function for the chosen bar.
- This kind of cooperation is called `delegation`. Each class does the part of the job that lies within its expertise, and lets the other class do the rest.
- `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. A public class with a private helper class is a common design pattern for containers where one container holds many items.

- 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.
- Row::head points to the first cell of the linked list.
- A Cell is created is created, initialized, and inserted into linked list in one line!

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.