Chapter 8: Interacting Classes

8.1 The Roles of a Class

Three more basic OO design principles:

> **A class is the expert on its own members.**
> **Oil and water don't mix.**
> **Delegate! Simplify!**

In this chapter we begin to build complexity by introducing classes that interact, or `collaborate` with each other to get the job done. Classes serve a variety of purposes in OO programming, and we try to define our classes so that they serve these purposes well. A class is...

- A type from which objects (instances) can be formed. We say the instances *belong to* the class.
- A collection of things that belong together; a struct with its associated functions.
- A way to encapsulate behavior: a public interface with a private implementation.
- A way to protect the integrity of data, providing the rest of the world with functions that provide a view of the data but no way to modify it.
- A way to organize and automate allocation, initialization, and deallocation
- A reusable module.
- A way to break a complex problem down into manageable, semi-independent pieces, each with a defined interface.
- An entity that can collaborate with other classes to perform a task.

8.1.1 Finding the Classes

In designing an OO application, we must first identify the classes that we need, based on the problem specification. This is more of an art than a science and experience plays a large role in making good choices. For a beginner, it helps to have some idea of the kind of classes that others have defined in many kinds of OO applications. These include:

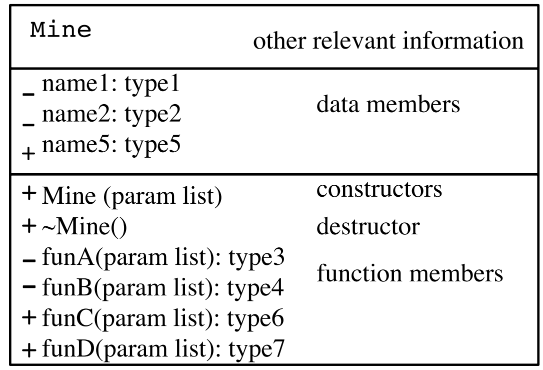
- Physical or tangible objects or places: airplane, airport, document, graph
- Non-tangible objects or systems: creditAuthorization, authorizationSystem
- Specifications for, or records of, objects: floorPlan, receipt
- Events or transactions: sale, charge, reservation, crash
- Organizations: csDepartment
- Roles of people: administrator, clerk
- Containers of things: jobList, facultySet, partsInventory, cardCatalog
- Things that go into the container: job, faculty, part, card

8.1.2 Diagramming One Class

A class is represented by a rectangle with three (or, optionally four) layers, as shown on the right:

- class name
- data members (type and name)
- function members (prototypes)
- other relevant information: friends, etc

Members are marked with + for public, # for protected, or – for private. In simplified diagrams, some of the layers may be omitted.



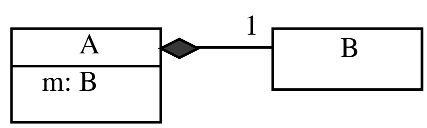
8.2 Class Relationships

Several relationships may exist between two classes: composition, aggregation, association, derivation, and friendship. These are explained in the paragraphs below and illustrated by class relationship diagrams. The four relationships at the top of this list provide the most protection for members of class B. As we move down the list, protection decreases, that is, class A functions get more privileges with respect to the private parts of B.

8.2.1 Composition.

Class A *composes* class B if a B object *is part of* every A object. The B member must be born when A is born, die when A dies, and be part of exactly one A object. Class A could compose any definite number of B objects; the number is indicated at the B end of the A-B link . The black diamond on the A end of the link indicates composition.

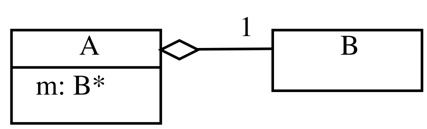
In this diagram, class A has one member named m of class B, which is constructed when an A object is constructed. When the A object is deallocated, it will cause the deallocation of the corresponding instance of B. A can use the public functions of B to operate on m.



8.2.2 Aggregation.

Class A *aggregates* class B if a B object *is part of* every A object, but the lifetime of the B object does not match the lifetime of the A object. The B object may be allocated later and attached to A with a pointer. Also, one B object might be aggregated by more than one A object, and A could aggregate more than one B object.

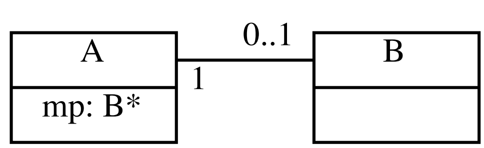
In the diagram, the white diamond on the A end of the A-B link indicates aggregation. Class A has one member named mp of class B*. When an instance of A is deallocated, it might or might not cause the deallocation of the corresponding instance(s) of B. A can use the public functions of B by using mp->.



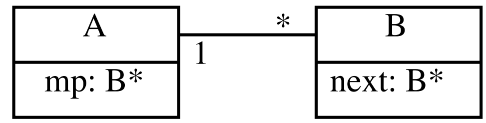
8.2.3 Association.

Class A is *associated* with class B if a member of A points at an instance of class B (or a set of instances), but both kinds of objects have an independent existence. Neither one is "part of " the other. The B object is generally not created by A's constructor; it is created independently and attached to A.

Simple Association. In a simple association, class A is associated with a definite number of B objects, often one. This is usually implemented in A by one pointer member that may be NULL or may be connected to an object or array of objects of type B. A can use the public functions of B by using mp->.

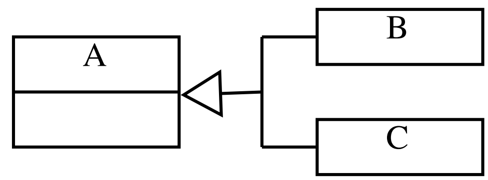


One-many Association. Class A is associated with a set of objects of class B. This set may be ordered or not. The relationship can be implemented by a member of A that points at a data structure containing objects of type B. In this case, A can use the public functions of B by using `mp->`. Alternatively, using STL classes, a member of A is an iterator for the set of B objects.



8.2.4 Derivation.

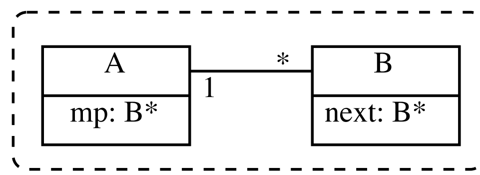
Class B can be *derived from* class A, which means that a B object is one possible variety of type A. Using derivation, a single object may have more than one type at the same time. Every instance of B or C is also an instance of A; it has all the properties of A, and inherits all of A's functions. The derived classes will extend either the set of A's properties, or functions, or both, and B and C will extend A differently.



In the diagram, the triangle with its point toward A indicates that classes B and C are derived from class A. We will study derivation after the midterm.

8.2.5 Friendship.

Class B can give *friendship* to A. This creates a tightly-coupled pair of classes in which the functions of A can use the private parts of B objects. This relationship is used primarily for linked data structures, where A is a container for items of type B and implements the interface for both classes. All members of class B should be private so that outside classes must use the interface provided by A to access them. (An alternative is to declare the class B inside class A, with all members of B being public.)



In the diagram, we show a dashed "rubber band" around the two class boxes, indicating that each one relies on the other.

8.2.6 An example.

Suppose you wish to implement a family datebook:

- The Datebook will have an array of linked lists, one for each family member.
- Each List points at a Node.
- Each Node contains one appointment and a pointer to the next Node.
- One appointment can be shared by two or more lists.

The resulting data structure is illustrated in Figure 7.1. In this structure, the Datebook *composes* four Lists, each list has a one-many *association* with a set of Nodes, and each Node *aggregates* one Appointment. Finally, *friendship* is used to couple the List and Node classes because the List class will provide the only public interface for the Node class. The UML diagram in Figure 7.2 shows these relationships.

8.2.7 Elementary Design Principles

Several design issues arise frequently. We list some here, with guidelines for addressing them and the reasons behind the guidelines.

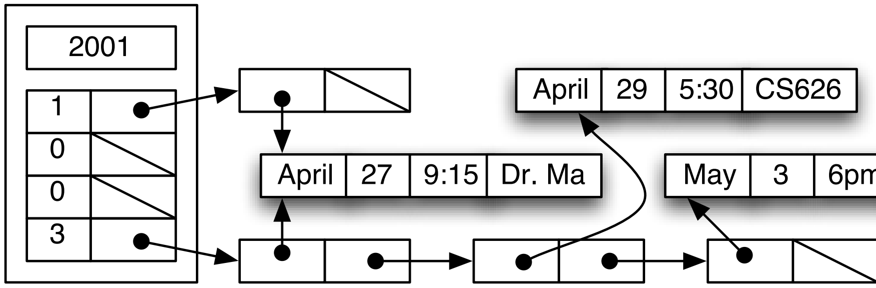


Figure 8.1: The data structure: an array of linked lists.

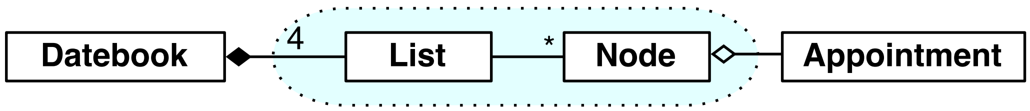


Figure 8.2: UML diagram for the Datebook data structure.

Privacy. Data members should be private. Public accessing functions should be defined only when absolutely necessary. [Why] This minimizes the possibility of getting inconsistent data in an object and minimizes the ways in which one class can depend on the representation of another. In the Bargraph program, later in this chapter, all data members of all classes are private.

Expert. Each class is its own expert and knows best what should be done with its members and how to do it. All functions that use data members should be class functions. If a class seems to need access to a member of another class in order to carry out an operation, it should delegate the operation to the class that is the expert on that data member.

There is one exception to this rule: To implement a linked structure, such as a List in the previous example, an auxiliary class is needed (Node). These two classes form a tightly coupled pair with an interface class (List) and a helper class (Node). A "friend class" declaration is used to create this relationship. All access to Nodes is through functions of the List class.

In the Bargraph program (later in this chapter) the two members of class Item are private. All access to Items is through the constuctor, destructor, and Print functions that belong to the class. However, in the tightly coupled pair of classes, Row and Cell, Row provides the interface for Cell, so Cell gives friendship to Row, allowing Row to set, change, and print the members of Cell.

Creation. The class that composes or aggregates an object should create it. [Why] This minimizes the ways in which one class depends on another (coupling).

In the Stack class, stack allocates storage for the objects (characters) stored in it. When that storage becomes full, the class allocates more space. In the Bargraph program, the Row class allocates space for Cells because the Row class contains the list of cells. It allocates the Items also, because Cell is a subsidiary class for which Row supplies the interface.

Deletion. The class that aggregates an object should generally delete it. [Why] To minimize confusion, and because nothing else is usually possible.

The array that was allocated by the Stack constructor and aggregated by the Stack class is freed by the Stack destructor. Also, in the push() function, each time the stack "grows", the old storage array is freed as soon as the data is copied into the new allocation area. In the Bargraph program, allocation happens at two levels. The Graph constructor allocates and aggregates 11 Rows, and the Graph destructor deletes them. The function Row::Insert allocates an Item and a Cell, which are then associated with the Row. At termination, the Row destructor deletes all the Items and all the Cells associated with it.

Consistency. Only class functions should modify class data members. There should be no "set" functions that allow outsiders to freely change the values of individual data members. [Why] To ensure that the object always remains in a consistent state.

Consider a class such as Row, that implements a linked list. All actions that modify the list structure should be done by Row class functions. The only class that should even know about tis structure is Row, itself. Under no circumstances should the Row class provide a function that lets another class change the pointer in a Cell. If another class is allowed to change list pointers, the list may be damaged beyond recovery.

Delegation. If a class A contains or aggregates a data member that belongs to class B, actions involving the parts of that member should be delegated to functions in class B. [Why] To minimize dependency between classes.

In the Bargraph program, each public class has its own print function. The Graph class delegates to the Row class the job of printing each Bar. That class, in turn, lets the Item class print the data fields of each Item.

Don't talk to strangers. Avoid calling functions indirectly through a chain of references or pointers. Call functions only using your own class members and parameters and let the member or parameter perform the task by delegating it in whatever way makes local sense. [Why] To avoid and minimize the number of indirect effects that happen when a class definition changes.

In the Bargraph program, the Graph class aggregates the Row class, the Row class associates with the Cell class and aggregates the Item class. So the Graph class is free to call Row functions, but it should not be calling Cell functions or Item functions. To do a Cell or Item operation, Graph should delegate the task by calling a Row function (and Row should provide functions for all such legitimate purposes.)

Responsibility. Every class should "take care of" itself, validate its own data, and handle its own emergencies. [Why] To minimize coupling between classes.

In the Stack program, the Stack checks whether it is full and if it is, it allocates more storage. There is no reason that a client program should ever ask about or be aware of a "full" condition.

8.3 The Program: Making a Bar Graph

8.3.1 Specification

Scope: Make a bar graph showing the distribution of student exam scores.

Input: A file, with one data set per line. Each data set consists of three initials and one exam score, in that order. (You may assume that all lines have valid data.) A sample input file is shown below, with its corresponding output.

Requirements: Create and initialize an array of eleven rows, where each row is initially empty but will eventually contain a linked list of scores in the specified range. Then read the file and process it one line at a time. For each line of data, dynamically allocate storage for the data and insert it into the appropriate row of scores.

Do this efficiently. When adding an item to a row, compute the subscript of the correct row. Do not identify it using a switch or a sequential search. Also, do not traverse that row's list of entries to find the end of the list; insert the new data at the head of the list. This is appropriate because the items in a list may be in any order.

Formulas: Scores 0...9 should be attached to the list in array slot 0; scores 10..19 should go on the list in slot 1, etc. Scores below 0 or above 99 should go on the last list. The order of the items in each row does not matter.

Output: A bar graph with 11 horizontal bars, as shown below.

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 :

```
Put input files in same directory as the executable code.
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
```

8.3.2 The Main Program and Output

```
1  //===========================================================================
2  // Bargraph of Exam Scores: An introduction to class interactions.
3  // Read scores from a user-specified file; display them as a bar graph.
4  // A. Fischer, September 30, 2000                           file: graphM.cpp
5  // Modified M. & A. Fischer, September 17, 2009
6  //===========================================================================
7  #include "tools.hpp"
8  #include "graph.hpp"
9
10 int main ( void )
11 {
12     banner();
13
14     Graph::instructions();          // Call of static class method
15     char fname [80];
16     cout << "Name of data file: ";
17     cin >> fname;                   // *** IRRESPONSIBLE CODE ***
18     // cin.getline(fname, 80);      // Prevents buffer overrun
19     ifstream infile (fname);        // Declare and open input stream
20     if (!infile)  fatal( "Cannot open %s for input - aborting!!\n", fname );
21     else say ( "File is open and ready to read.\n");
22
23     Graph curve( infile );          // Declare and construct a Graph object.
24                                     // Realizes data structure from a file
25     cout <<curve;                   // Print the graph.
26     // OR: curve.print( cout );     // Print the graph.
27
28     bye();
29     // Storage belonging to curve will be freed by destructors at this time.
30 }
```

Notes on the main program. The main program forms the interface between the user and the top-level class, Graph. It gives instructions (line 14) sets up an input stream for Graph's use and gives appropriate feedback (lines 15…21), then calls the Graph constructor (line 23) and print function (line 25) to construct and print a bargraph. Although four classes are used to implement this program, three of them are subsidiary to Graph. No other classes are known to or used by `main()`. Thus, the only class header file included by this module is `"graph.hpp"`.

The output. The output consists of three sections. First are the banner, instructions and input prompt printed by main. Following this is the normal program output (a bar graph, exactly like the one shown in the specifications). Last is the trace that is printed by the class destructors. By including trace output in your destructors, you can check that all parts of your data structures are freed properly and prove that you have no memory leaks.

Below, we give the first and last parts of the trace to illustrate the process of memory deallocation. (The dotted line marks the part of the trace that was omitted.) The lines printed by the Graph and destructors are not indented, those printed by the Item destructor are indented, and the Cell destructor prints one word on the same line as the Item destructor. Rows and Items tell us which instance is being deleted. Note that the parts of a data structure are freed before the whole, that is, the Item before the Cell, the Cells before the Row, and the Rows before the Graph.

Here is the beginning and end of the program termination trace:

```
Normal termination.
   Deleting Item AWF ...  Cell
 Deleting row 00..09:   >>>
 Deleting row 10..19:   >>>
 Deleting row 20..29:   >>>
    Deleting Item PLK ...  Cell
 Deleting row 30..39:   >>>
 Deleting row 40..49:   >>>
    Deleting Item ABA ...  Cell
```

```
   ............................
Deleting row 80..89:   >>>
   Deleting Item GBS ...  Cell
   Deleting Item MJF ...  Cell
Deleting row 90..99:   >>>
   Deleting Item ALA ...  Cell
   Deleting Item JBK ...  Cell
Deleting row Errors:   >>>
Deleting Graph
```

8.3.3 The Data Structure and UML Diagrams

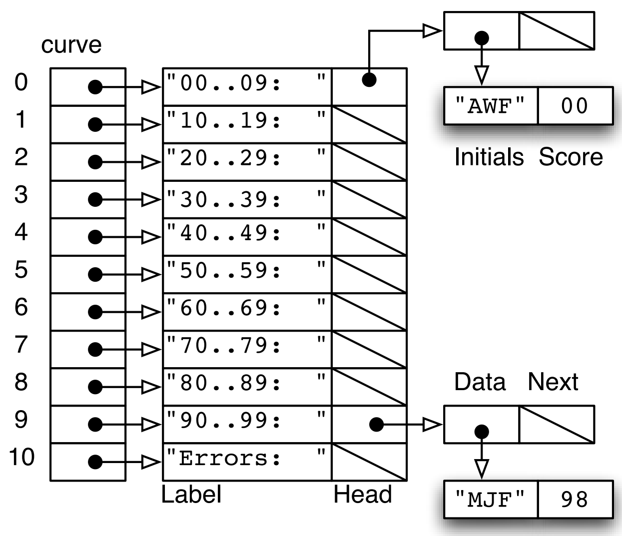


Figure 8.3: The graph data structure, after inserting two scores.

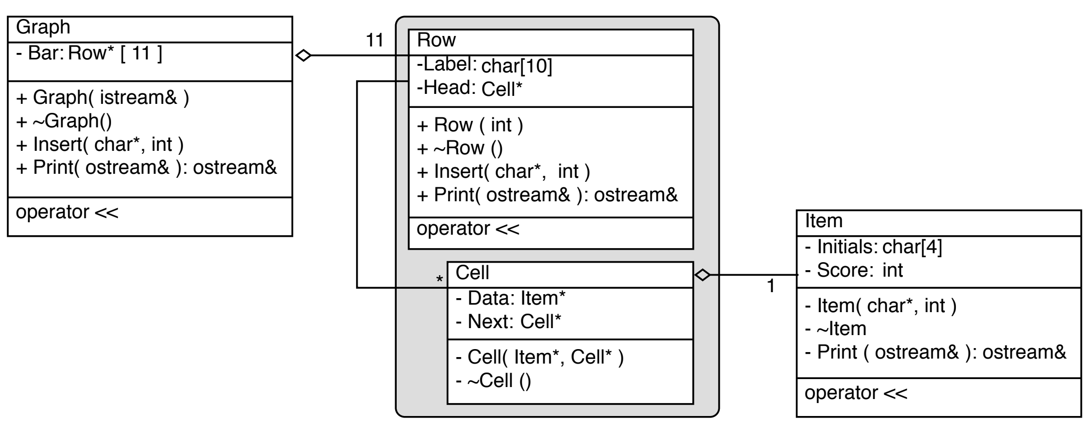


Figure 8.4: UML class diagram for Bargraph program.

Class relationships.

1. Part of initializing the Graph class is to allocate eleven new Rows and attach them to the backbone array (Bar). Since these are dynamically allocated, the relationship is not composition. But because there are exactly eleven Rows and their lifetime will end when the Graph dies, this is aggregation (not association).
2. Classes Row and Cell are friends, with Row providing the interface for both classes and Cell serving as a helper class for Row.
3. One Row contains an indefinite number of cells (zero or more), all allocated and deallocated at different times. This is association.
4. Each Cell contains a pointer to an Item. The Item is created by the Cell constructor and dies with the cell. This is aggregation. It is not composition because the Item is not physically part of the Cell (it is connected to the Cell by a pointer).

8.3.4 Class Item

```
//===========================================================================
// Item: A student's initials and one exam score.
// A. Fischer, October 1, 2000                               file: item.hpp
// Modified M. & A. Fischer, September 17, 2009
//===========================================================================
#pragma once
#include <iostream>
#include <string.h>
using namespace std;
//---------------------------------------------------------------------------
// Data class to model a student exam score
// Alice's design pattern:
//     oil and water don't mix -- keep data and data structure separate
// Constructor and destructor contain tracing printouts for debugging
class Item                      // One name-score pair
{
  private:                      // Variable names are private
    char initials [4];          // Array of char for student name
    int  score;                 // Integer to hold score

  public:
    inline Item (char* inits, int sc);
    ~Item (){ cerr <<"    Deleting Item " <<initials <<" ...\n"; }
    ostream& print ( ostream& os );
};

//---------------------------------------------------------------------------
// Inline constructor, defined outside of the class but in header file
// Precondition: strlen(inits) <= 3
Item::Item (char* inits, int sc){
    strcpy( initials, inits );
    score = sc;
}

//---------------------------------------------------------------------------
inline ostream&    // inline can be declared here or within class (or both)
Item::print ( ostream& os ){
    return os <<initials <<" " <<score <<" ";
}

// Extend global output operator << -----------------------------------------
// Item::print() is expert at printing Item data
inline ostream&
operator << (ostream& out, Item& x){ return x.print( out ); }
```

We consider this class first because it is the simplest and relates to only one other class.

- The four functions defined here should be provided for every class: a constructor, a destructor, a print function, and an operator extension for `<<`.
- Every class function has a class object as its implied parameter, in addition to the explicit parameters listed in the parentheses. Thus, the Item constructor has three parameters (one implied and two explicit). We rarely refer to the implied parameter as a whole, but when we use the names of class members, we mean the members of the implied parameter.
- This class is defined entirely within the .hpp file because there are only three class functions and they are all short. The destructor fits entirely on one line, so it is given on line 49. The other two functions are declared immediately after the right-brace that closes the class declaration (lines 53...65). Both are "inline". Item is declared "inline" on line 48; print is declared "inline" on line 62. Better style would have the word "inline" on both the declaration and the later definition.
- Lines 62–65 extend the `<<` operator to work on Items by defining it in terms of Item::Print(). Some books do this by declaring a friend function. However, friend declarations should be avoided wherever possible because they give free access to all class members. The technique shown here lets us extend the global operator without making data members public and without using a "friend function".
- The Print function returns the stream handle it received as an argument. This makes it easier to extend the `<<` operator for the new class. The operator `<<` *must* return an `ostream&` result to conform to the built-in definitions of `<<` and to make it possible to chain several `<<` operators in one expression. The definition given for `Print` returns an `ostream&` result to make it work smoothly with the operator extension given for `<<`. Both local definitions could be changed, as shown below, but they need to be paired correctly: the second definition of `Print` won't work with the first definition of `<<`.

```
void Print ( ostream& os ){ os <<Initials <<" " <<Score <<" "; }
inline ostream& operator << (ostream& os, Item& T){ T.Print( os );  return os; }
```

- The C++ language designers defined `<<` as a global (non-class) function so that it could be easily extended for every new class. Even though it works closely with the Item class, the operator extension does not belong to that class because `<<` is a global function. Therefore, the extension must be written outside the class.
- The extension of `<<` is written within the .hpp file for a class so that it will be included, with no extra effort, wherever the Item class is used. It must be declared "inline" to avoid linker errors due to multiple inclusion. This is a confusing issue that we will defer to later. For now, copy this pattern.
- The class definition file has a minimal header; it gives the author's name, last modification date, and the name of the file. In a commercial setting, a more extensive header would be required.
- The preprocessor command, `#pragma once` is used routinely in header files to prevent the same declarations from being included twice in the same compile module and to prevent circular `#includes`.

8.3.5 Class Graph

Notes on the Graph declaration.

- The class declaration starts and ends with the usual preprocessor commands. We include the header file for the Row class because a graph is built of Rows. We do not need the header for the Item class because the Graph functions do not deal directly with Items; all actions are mediated by the Row class.
- A Graph has only one data member, an array of pointers to Rows. Since the number of bars is known at compile time, it would be possible to declare a Graph as an array of Rows, not an array of Row*. Composing Rows is more efficient, but aggregating Row*s is more flexible. The Row* implementation was chosen so that the Row constructor could be called eleven times, to construct eleven rows, with the row number as an argument each time. This lets each Row be initialized with a different label. If an array of Row were used, all rows would need to be initialized identically.
- The constructor and destructor for this class are public because the main program will declare (create) an instance of Graph named "curve". The Graph destructor will be called automatically, from main, after the call on bye(), when curve goes out-of-scope.

- The argument to the constructor is the stream that contains the data, which should already be open. The argument is passed by reference for two reasons. First, call by value will not work here because reading from a stream changes it. Second, the stream is an object with dynamic extensions. All such objects are passed by reference or pointer to avoid the problem of shallow-copy and deep-deletion.
- Since all calls on Insert are within the Graph class, it is defined to be a private function. However, it is quite reasonable that someone might want to reuse this class in a program that does interactive input as well as file input. In that case, we would need to make the Insert function public.
- The extension of `operator<<` is included in the class header file because it is used as if it were a class function. However, it cannot be an ordinary class function because its first parameter is an ostream&, not a Graph. Being outside the class, the definition of `operator<<` cannot access the private parts of the class. It does its job by calling Graph::Print, which is a public function. Print, in turn, accesses the private class members. This is a typical illustration of how the privacy of class members can be maintained.

```
//===========================================================================
// Declarations for a bar graph.
// A. Fischer, April 22, 2000                                   file: graph.hpp
// Modified M. & A. Fischer, September 17, 2009
//===========================================================================
#pragma once
#include "tools.hpp"
#include "row.hpp"
//#include "rowNest.hpp"  // alternative way to define a recursive type
#define BARS 11

//---------------------------------------------------------------------------
class Graph {
  private:
    Row* bar[BARS]; // Each list is one bar of the graph.  (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 ); }
```

```
//==========================================================================
// Implementation of the Graph class.
// A. Fischer, April 22, 2000                                  file: graph.cpp
// Modified M. & A. Fischer, September 17, 2009
//==========================================================================
#include "graph.hpp"
//-------------------------------------- Use an input file to build a graph.
Graph::Graph( istream& infile ){
    char initials [4];
    int score;
    for (int k=0; k<BARS; ++k) bar[k] = new Row(k);
    for (;;){
        infile >> ws;               // Skip leading whitespace before get.
        infile.get(initials, 4, ' ');  // Read three initials ... safely.
        if (infile.eof()) break;
        infile >> score;            // No need to skip ws before using >>.
        insert (initials, score);   // *** POTENTIAL INFINITE LOOP ***
    }
}

```

```
// -------------------------------------------------------- Delete all Rows.
Graph::~Graph() {
    for (int k=0; k<BARS; ++k)  delete bar[k];
    cerr <<" Deleting Graph\n";
}

// ----------------------------------------------- Insert a node into a Row.
void
Graph::insert( char *initials, int score ){ // Function is private within class.
    const int intervals = BARS-1;
    int index;
    // Calculate insertion index for score
    if (score >= 0 && score < 100)      // If score is between 0-99, it
        index = (score / intervals);    // belongs in one of first BARS-1 rows.
    else
        index = BARS-1;                 // Errors are displayed on last row
    bar[index]->insert( initials, score );  // delegation
}

// --------------------------------------------- Print the entire bar graph.
ostream&
Graph::print( ostream& out ){
    out << "\n";
    for (int k=0; k<BARS; ++k)
        out << *bar[k] <<"\n";          // Delegate to Row::print()
    return out;
}
```

Notes on the Graph implementation. We put the definitions of all four class functions in a separate .cpp file because all are longer than a line or two.

- After allocating eleven empty Rows, the Graph constructor reads lines from the data file and calls the Graph::Insert function to store the data. Three lines of code are used to read each line of data because one of the data fields is a string. When `>>` is used to read a string, there is no protection against long inputs that overflow the bounds of the data array and start destroying other values in memory. For this reason, we use `get` rather than `>>` to read all strings. It is necessary to skip whitespace before the call on `get`, and calls on `get` and `>>` cannot be mixed in the same statement. Thus, three lines of code are required. If we do not care whether the input is done "safely" or not, these three lines could be condensed into one:

  ```
  infile >> Inits >> Score;
  ```

- Eleven Rows are created by the Graph constructor, so the Graph destructor contains a loop that deletes eleven Rows.
- The Insert function decides which row the new data belongs in, then delegates the insertion task to the appropriate Row. This is a typical control pattern; as smaller and smaller portions of the data structure are used, control passes from each class to the class it contains.
- Expertise: The Graph class is an expert on the meaning of its rows, so it selects the row to be used. The Row class is the expert on how to store data in a Row, so it accepts the data, creates a storage cell, and does the insertion.

8.3.6 Classes Row and Cell

Notes on the Cell class declaration.

- Together, the Cell and Row classes implement a linked list in which any number of data items can be stored. A Cell consists of a pointer to an Item and a pointer to the next Cell.
- The Row and Cell class declarations have been placed in a single .hpp file because they form one conceptual unit. Each class depends on the other and neither makes sense alone.

- Friendship. The Cell class is a private class (all of its members are private). It is used only to implement the Row class. It gives friendship to Row, which implements the interface for both classes. The private class (Cell) comes first, because the interface class (Row) refers to it. No way of any kind is provided for other parts of the program to access Cells.
- Each Cell aggregates one Item, which holds the data for one student. On the UML diagram, Cell and Item are connected by a line with a white aggregation diagram at the Cell end and an optional "1" at the Item end. As with Graph and Row, composition could be used, rather than aggregation, to put the Item in the Cell. In that case, the type of Data would be Item, not Item*. Aggregation is less efficient because it involves an extra layer of pointers, but it permits us to construct Items that are independent of the Cells. These items can be constructed and used in other parts of the program that have no relation to Cells or Rows.
- The Cell class constructor and destructor are short and routine and are defined inline. They, also, are private because they will be used only by Row functions. No other part of the program has any reason to create or destroy a Cell.

```
//===========================================================================
// Class for a linked-list row and its cells
// A. Fischer, October 1, 2000                                  file: row.hpp
// Modified M. & A. Fischer, September 17, 2009
//===========================================================================
#pragma once
#include <iostream>
#include "item.hpp"
using namespace std;
//---------------------------------------------------------------------------
// Dependent class.  Holds an Item and a link to another Cell
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"; }
};

//---------------------------------------------------------------------------
// Data structure class
class Row   {   // Interface class for one bar of the bargraph.
  private:
    char label[10];             // Row header label
    Cell* head;                 // Pointer to head of row

  public:
    Row ( int n );
    ~Row ();
    void insert ( char* name, int score );  // delegation
    ostream& print ( ostream& os );
};

//---------------------------------------------------------------------------
inline ostream& operator << ( ostream& out, Row& T){ return T.print( out ); }
```

The Row class declaration.

- Row has a One-to-Many association with Cell, denoted in the UML by a "1" at the Row end and a "*" at the Cell end. Together, these classes form a classical container class for Items.
- Unlike a typical C or Pascal implementation of a linked list, the type Row is not the same as the type Cell*. The head of the linked list of Cells is one (but only one) of the members of Row. Other data

members include scanners and functions for processing lists and, in this example, an output label for the Row.

- The constructor and destructor are not defined inline because they are long and contain loops. The destructor for a linked-list class needs to free all the storage allocated by the constructor and by other class functions that add items to the linked list.
- As always, a Print function (and a corresponding extension for `<<`) are defined for Rows. No such functions are defined for Cell because Row provides the interface for both classes.
- Many times, a container class is used to implement an active database, so it has functions to Insert, Delete, Modify, and Lookup items. In this program, the container is used as a way to sort exam scores, and needs only one database function: insert an Item.

Notes on the Row class implementation. All of the functions in the `row.hpp` file belong to the Row class because the Cell functions were all defined inline. This code file is the heart of the program.

- The Row constructor creates an output label for the row and initializes the pointer, Head, that will point at the list of Item Cells. Head is set to NULL because, at this time, the list is empty. A scanning pointer, Curr, is also set to NULL but could be left uninitialized.
- Following the expert pattern, the Graph::Insert function calls the Row::Insert function to do the actual insertion. In general, this is the way operations should be implemented: the task is handed down the line, from the user interface to the primary class, and from class to class below that, until it reaches a low-level class that knows how to carry out the task.

```
//===========================================================================
// Implementation of class Row.
// A. Fischer, April 22, 2000                                    file: row.cpp
// Modified M. & A. Fischer, September 17, 2009
//===========================================================================
#include "row.hpp"
//---------------------------------------------------------------------------
// Row number is used to construct a label for the row
Row::Row( int rowNum ){
    if (rowNum == 10) strcpy( label, "Errors:  " );
    else {
        strcpy( label, " 0.. 9:  " );
        label[0] = label[4] = '0'+ rowNum;  // example: label=="70..79"
    }
    head = NULL;  // represents empty list of data
}
//---------------------------------------------------------------------------
// Row is responsible for deleting everything created by this class
Row::~Row(){
    Cell* curr;
    while (head != NULL){
        curr=head;
        head=head->next;
        delete curr;
    }
    cerr << " Deleting row " << label <<" >>> \n";
}
//---------------------------------------------------------------------------
// Create and insert Cell into linked list at head
// Design pattern: creator.  Item is created by Cell constructor.
void
Row::insert( char* name, int score ){
    head = new Cell( name, score, head );  // put new cell at head of list
}
//---------------------------------------------------------------------------
// Design decision: print Cell data directly; no delegation of print
ostream&
```

```
Row::print( ostream& os ){
    Cell* curr;
    os << label;
    for (curr=head; curr!=NULL; curr=curr->next)
        curr->data->print( os );    //  OR: os << *(curr->data);
    return os;
}
```

- Row::Insert does not pass the insertion command on to Cell::Insert because Row is the lowest level class that is "expert" on multiple Cells. The Row class allocates a new Cell and passes the data to it because the Cell is the "expert" on Items. The Cell constructor finally creates an item and stores its pointer. In C, eight routine and tedious lines of code are needed to to the allocation and initialization:

```
Item* tempI = (Item*) malloc( sizeof(Item) );
strncpy( tempI->Initials, 4, name );
tempI->Score = score;
Cell* tempC = (Cell*) malloc( sizeof(Cell) );
tempC->Data = tempI;
tempC->Next = Head;
Head = tempC;
```

  A good demonstration of the power and convenience of C++ constructors is that they permit us to do the same job in one line: `Head = new Cell( new Item(name, score), Head );`

  In this program we do part of the construction in the Row class and part in the Cell class, following the principles of Creator and Expert:

```
In Row:    Head = new Cell( name,  score, Head );
In Cell:   Data = new Item( name, score );
```

- The Print function loops through the Cells on the linked list and sends a Print command to the Item in each cell. Arrows are used in this command (rather than dots) because each Item is attached to the Cell, and each Cell is attached to the Row through a pointer.
- The Row destructor uses a loop to delete, or free, all of the memory blocks allocated by Row::Insert. When it deletes each Cell, the Cell destructor will delete the Item it contains.

8.4 An Event Trace

In one common programming pattern, a program has five phases of operation:

- During the setup phase, the main program acquires resources and creates its primary data structure(s).
- During the input phase, files and interactive entry are used to store data in that structure.
- Processing accesses and modifies the information.
- The output phase prints an analysis of it.
- Finally, during the cleanup phase, the data is written back onto some permanent storage medium, streams are closed, and storage is freed.

In C, and even more in C++, some phases are often combined. For example, setup can be partially merged with input, and processing can be merged with either input or output. In the Bargraph program, there is no processing step. This leaves four major phases: setup, input, output, and cleanup. The event traces in the next four diagrams illustrate these four phases of the program's operation.

Interpreting a trace. An event trace provides a dynamic 2-dimensional view of the phases of program execution. It shows the order in which things are created and freed and how control passes back and forth between the classes. Figures 7.5...7.8 show the event trace for the bargraph program.

- **The columns.** An event trace has dashed vertical lines. Each represents the "territory" of one program module: either main() or one of the classes used by the program.

- **Passing control.** Arrows represent the flow of control. When an arrow stops at a vertical line, control enters that class. The label on an arrow indicate the function call that caused control to leave one class and enter another.
- **Loops.** Loops are indicated by large ovals and are documented informally.

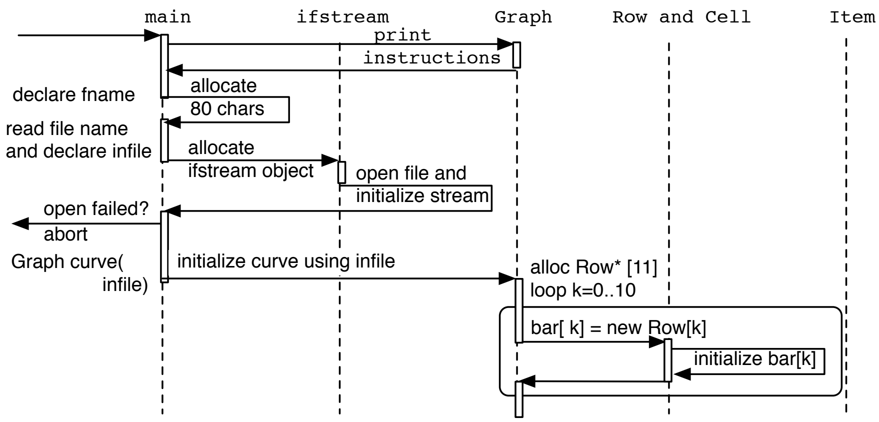


Figure 8.5: Event trace phase 1–Constructing the Graph.

- **The passage of time.** When the program begins, the system sends control into main() at the upper left corner of the graph. As long as control remains within the same class, time flows downward along its dotted line. A broad white time line indicates that control is within a class. Solid arrows represent function calls. As time passes, they take control from one class to another and also lead down the page.

Phase 1: Setup. During this phase (Figure 7.5), the main program interacts with the user to set up the input stream. Then two major data structures are declared and initialized: an istream named `input` and a Graph named `curve`. When curve is declared, the system creates storage for an array of eleven pointers, then calls the Graph constructor to initialize that array and construct its extensions.

During construction, space is allocated for the eleven rows of the graph. As each Row is created, the Row constructor is called to initialize it. (It creates a label for the row and sets the list pointers to NULL.)

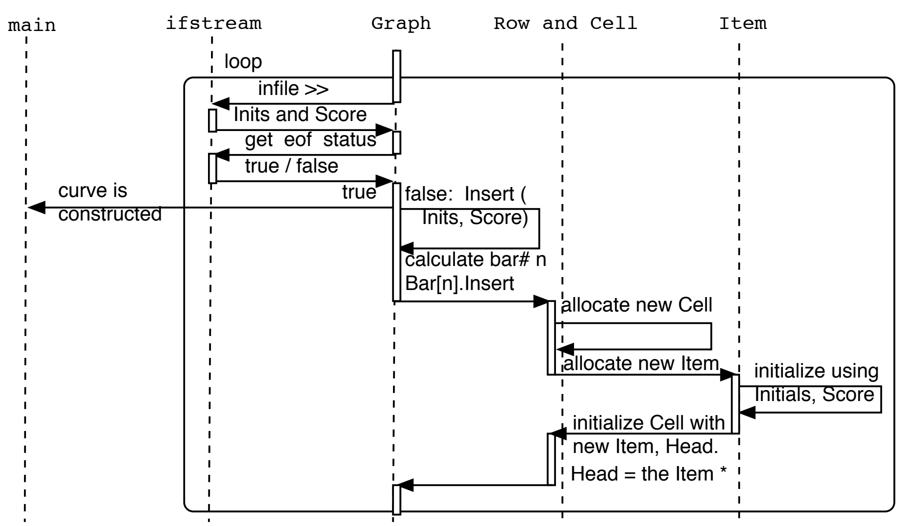


Figure 8.6: Phase 2, Filling the Graph.

Phase 2: Input. A loop is used to read data lines from the input stream. For each, storage is allocated, and the resulting structures are linked into the appropriate Row, at the head of its list of Cells. The new storage is

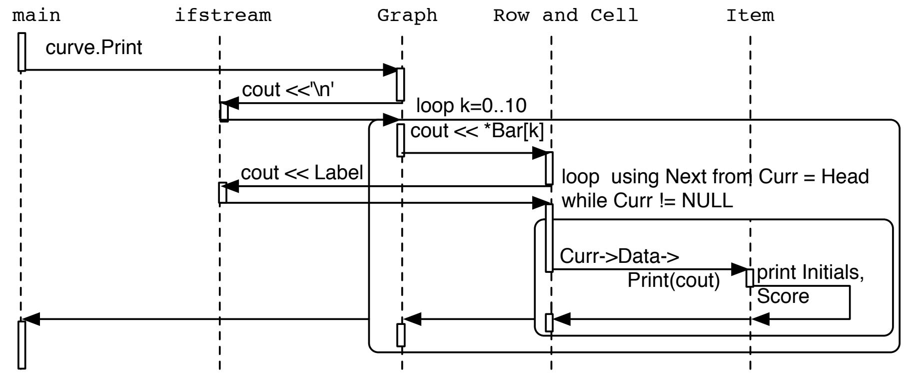


Figure 8.7: Phase 3, Output.

initialized by the constructors in the Row, Cell, and Item classes. (Figure 7.6) Nothing is deleted during this process because no extra or temporary locations are created. We are building a database; every object allocated gets connected to it and used later in the program.

Phase 3: Output. The purpose of this program is to organize the data into a meaningful format, a bar graph. The organization is done by the constructors during the input phase. No other processing is required, so printing the graph is the next step (Figure 7.7). Most programs print header and footer titles. Here, none are required by the specifications, so we print the graph by simply printing each of its 11 rows. To do so, we use the extended << operator, which calls on Row::print() (the expert on displaying rows) to do the job.

After Row::print() prints the row labels, it must print the data. Since Item::print() is the expert on how an Item should look, Row::print() calls it to print the Item in each Cell, starting with the Head cell and ending with NULL.

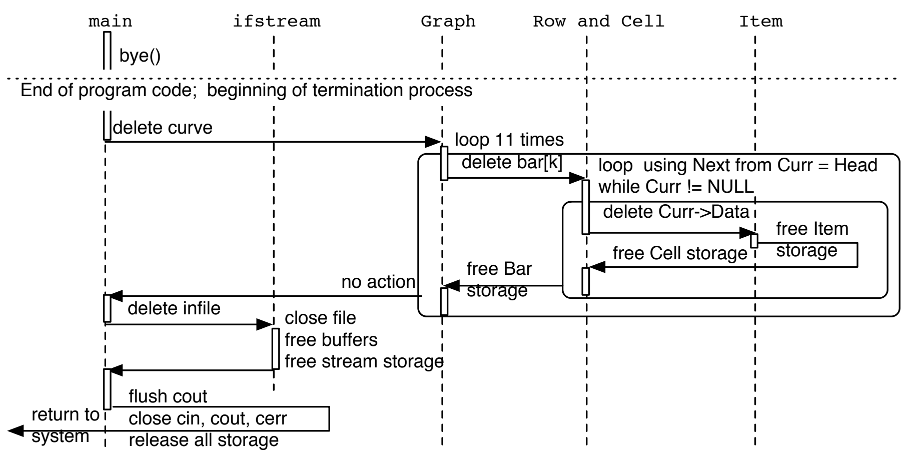


Figure 8.8: Phase 4, Deallocation.

Phase 4: Deallocation. When the data in the primary data structure has been created or modified during execution, it must usually be written to a file at the end. It is appropriate to have the destructors write the output file during the process of freeing the memory. This has the nice property that the data resides in exactly one place at any time; there is always one copy of it, not two, not zero.

In this program, no data was created during the run, so no output file is needed. The destructors are therefore simple: they call delete for each pointer that was created by new, and finish by printing trace comments.

Declared objects are deleted automatically in the order opposite their creation order (Figure 7.8). Thus, the Graph is deleted, then the istream, then the character array. The trace shows calls on destructors for the class objects.