Chapter 15: Polymorphism and Virtual Functions

From Lewis Carrol, *Through the Looking Glass*:

> "When I use a word," Humpty Dumpty said, in rather a scornful tone, "it means just what I choose it to mean – neither more nor less."
>
> "The question is," said Alice, "whether you can make words mean so many different things."
>
> "The question is," said Humpty Dumpty, "which is to be master– that's all."

15.1 Basic Concepts

15.1.1 Definitions

Simple derivation. The Hangman program uses the simple form of derivation: two application classes are derived from the same base class. This kind of derivation serves two important purposes:

- To factor out the common parts of related classes so that the common code does not need to be written twice. An example of this is the BaseWord class with its derivatives, Alphabet, and Hangword.
- To facilitate reuse of library classes and templates. The base class will contain a generally-useful data structure and its functions. The derived class will contain functions that are specific to the application. An example of this is the RandString class, which is derived from an instantiation of the FlexArray template.

Polymorphic derivation. In this chapter, we introduce virtual functions and two complex and powerful uses for derived classes that virtual functions support: abstraction and polymorphism.

- A *virtual function* is a function in a base class that forms part of the interface for a set of derived classes. It is declared *virtual* in the base class and may or may not have a definition in that class. It *will* have definitions in one or more of the derived classes. The purpose of a virtual function is to have one name, one prototype, and more than one definition so that the function's behavior can be appropriate for each of the derived classes.
- A *pure virtual function* is a function that has no definition in the base class.
- An *abstract class* is a class with one or more pure vitual functions.
- A *polymorphic* class is a base class that supports a declared set of public virtual functions, together with two or more derived classes that define methods for those functions. Data members and non-virtual functions may be defined both in the base class and the derived classes, as appropriate. We say that the derived classes *implement* the polymorphic *interface* class.

15.1.2 Virtual functions.

A virtual function is shared by the classes in a derivation hierarchy such as the Employee classes in Figure 15.1.2. (Note: In this sketch, three shapes have been drawn on top of the ordinary rectangular class boxes. This is not proper UML; the shapes will be used later to explain polymorphism.)

We create a virtual function when the same task must be done for all objects of all types in the hierarchy, but the method for doing this task depends on the particular representation of an object. For example, suppose the function calculatePay() must be defined for all employees, but the formula for the calculation is different for union members and professional staff. We want a function with one name that is implemented by three or more defining *methods*. For any given function call, we want the appropriate function to be called. For example, suppose we have declared four objects:

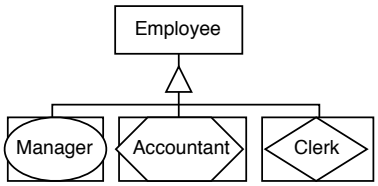

Figure 15.1: A polymorphic class hierarchy.

```
Manager M;
Accountant  A;
Clerk C, D;
```

Then if we call A.calculatePay() or M.calculatePay() we want the calculation to be made using the formula for professional staff. If we call C.calculatePay() or D.calculatePay(), we want the union formula used.

We implement this arrangement by declaring (and defining) calculatePay() as a virtual function in the Employee class, with or without a general method in that class. We also define calculatePay() in each of the derived classes, to make the appropriate calculations for the specific type of employee. The general method might do the parts of the calculation that are common to all classes. When calculatePay() is called, the system must select one of the methods to use. It does this by looking at the specific type of the object in the function call. This issue is explored more fully in the section on polymorphic classes.

Syntax. For examples, look at print() in Container, Linear and Queue, later in this chapter.

- The prototype for a virtual function starts with the keyword `virtual`. The rest of the prototype is just like any other function.
- The prototype for a pure virtual function ends in =0 instead of a semicolon. This means that it *has no definition* within the class.
- You must *not* write the keyword `virtual` before the definition of a virtual function that is outside the class.
- Any class with one or more virtual functions must have a virtual destructor.

How it works. Every object that could require dynamic dispatch must carry a type tag (one byte) at run time that identifies its relationship to the base class of the class hierarchy. (1st subclass, 2nd subclass, etc.) This is necessary if even one function is declared virtual, either in the class itself or in a parent class.

The run-time system will select the correct method to use for each call on a virtual function. To do so, it uses the type tag of the implied parameter to subscript the function's dispatch table. This table has one slot for each class derived from the class that contains the original virtual declaration. The value stored in slot k is the entry address for the method that should be used for the kth subclass derived from the base class. This is slightly slower than static binding because there is one extra memory reference for every call on every virtual function.

15.2 Polymorphic Classes

The purposes of polymorphism.

- To define an extensible set of representations for a class. One or two may be defined at first, more added later. It is a simple way to make an application extensible; you can build part of it now, part later, and be confident that the parts will work together smoothly.
- To allow a data structure to contain a mixture of items of different but related subtypes, such as the linked list of employees illustrated in Figure 15.2. The data objects in this list are the Employees defined in section 15.1.2; the shape of each object indicates which subtype of Employee it belongs to.
- To support run-time variability of types within a restricted set of related types, and the accompanying run-time dispatch (binding) of methods. The most specific appropriate method is dispatched. This lets us create different varieties of objects depending on input that is entered at run time.

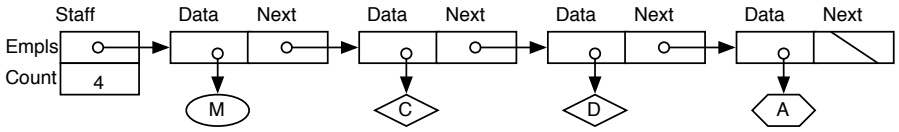

Figure 15.2: A list of polymorphic Employees.

Dispatching. Dispatching a function call is simple when the implied argument belongs to a derived class and a method for the virtual function is defined in that class. However, this is not always the case. For example, Figure 15.2 is a list of pointers to polymorphic Employees. Suppose a virtual print() function is defined in Employee and given a method there. Also, methods are defined in Accountant and Manager; these print the special data then call the Employee::print() to print the rest. However, no method is defined in Clerk because the general Employee method is appropriate. Now suppose we want to print the four Employees on the list and write this:

```
Employee* p;
for (p=Staff.Empls; p!=nullptr; p=p->Next) p->Data->print(cout);
```

When we execute this loop, the Employee::print() function will be *called* four times. If it were an ordinary function, Employee::print() would be *executed* four times. However, that does not happen because Employee::print() is virtual, which means that the most appropriate function in the class hierarchy will be executed when Employee::print() is called:

1. The system will dispatch Manager::print() to print M because M is a Manager.
2. To print C and D, Employee::print() will be used because the Clerk class does not have a print function of its own.
3. To print the Accountant, A, the system will dispatch Accountant::print().

In all cases, the most specific applicable method is selected at run time and dispatched. It cannot be done at compile time because three different methods must be used for the four Employees.

Two limitations. Virtual functions cannot be expanded inline because the correct method to apply is not known until run time, when the actual type of the implied parameter can be checked. For this reason, using a large number of virtual functions can increase execution time.

If you have a function that you want a derived class to inherit from its parent, do not define another function with the same name in the derived class, even if its parameter list is different.

Two design guidelines. If you have a polymorphic base class:

1. . . . and you allocate memory in a derived class, you must make the base-class destructor virtual.
2. You can use the C++ dynamic_cast operator for safe run-time polymorphic casting.

15.3 Creating a Polymorphic Container

Overview of the demo program. The major example in this chapter is a polymorphic implementation of linear containers. The class Container is an abstract class from which linear containers (lists, queues) and non-linear containers (trees, hash tables) could be derived. Container supplies a minimal generic interface for container clases (those that can be used to store collections of data items).

In this chapter and the next, we focus on linear containers. The class Linear is the base class for a polymorphic family of list-based containers. From it we derive several data structures: Stack and Queue in this chapter, List and Priority Queue in the next. All of these conform to the Container interface and Linear implementation strategy but present different insertion and/or deletion rules. We use the containers in this chapter to store objects of class Exam, consisting of a 3-letter name and an integer.

The classes Linear and Cell should be defined as templates so that they do not depend on the type of the data in the list. Templates were not used here because they would complicate the structure of the program and the important issue here is the structure of a polymorphic class. In a real application, both would be used.

What to look for.

- A virtual function can be initially declared with or without a definition. For example, pop() is declared without a definition in Container, but insert() is declared with a definition in Linear.
- A virtual function can be redefined (directly or indirectly) in a derived class, as Queue::insert() redefines Linear::insert(). When redefinition is used, the derived-class method is often defined in terms of the inherited method, as Stack::print() is defined in terms of Linear::print().
- If a function is declared virtual, then it is virtual in all derived classes.
- When a virtual function is executed on a member of the base class, the method defined for that function in the appropriate derived class is dispatched, if it exists. Thus, when insert() is called from Linear::put() to insert a new Cell into a Queue, Queue::insert() is dispatched, not Linear::insert(). If Linear::insert() were not virtual, Linear::insert() would be executed.

15.3.1 Container: An Abstract Class

```
1  // ---------------------------------------------------------------------
2  // Abstract Containers
3  // A. Fischer   June 10, 2001                          file: contain.hpp
4  // ---------------------------------------------------------------------
5  #ifndef CONTAIN_H
6  #define CONTAIN_H
7  #include "exam.hpp"
8  #include "cell.hpp"
9
10 class Container {
11   public:           // -----------------------------------------------------
12     virtual ~Container(){}
13     virtual void     put(Item*)      =0; // Put Item into the Container.
14     virtual Item*    pop()           =0; // Remove next Item from Container.
15     virtual Item*    peek()          =0; // Look but don't remove next Item.
16     virtual ostream& print(ostream&) =0; // Print all Items in Container.
17 };
18 #endif
```

A container is a place to store and retrieve data items of any sort which have been attached to cells. Each container has its own discipline for organizing the data that is stored in it. In this section we develop a generic linear container class from which special containers such as stacks or queues can be derived. We use the program to illustrate the concepts, syntax, and interactions among a polymorphic base class and its implementation classes.

The Container class presented here supplies implementation-independent functions that are appropriate for any container and any contents type. Every container must allow a client program to put data items into the container and get them back out. A print() function is often useful and is necessary for debugging.

15.3.2 Linear: A Polymorphic Class

A linear container is one that has a beginning and an end, and the data inside it is arranged in a one-dimensional manner. It might be sorted or unsorted. The class Linear defines a simple, unsorted, linear container that is implemented by a linked list, using a helper class named Cell. It defines a set of list-handling function that can be written in a generic way so that they will apply to most or all linked-list linear containers. All of these functions are protected except the public interface functions that were inherited from Container. The other functions are not public because they allow a caller to decide how and where to put things into or take things out of the list. This kind of control is necessary for defining Stack and Queue but not safe for public use.

Since Linear is derived from Container, it inherits all of the function prototypes of Container. Since these functions are equally appropriate for use with an array or a linked list, Linear could be implemented either way. We commit to a linked list implementation when we define the Linear constructor in line 37. The code for reset() (line 39), end() (line 40), and all the functions in the .cpp file also rely on a linked list representation for the data.

```
18  // ------------------------------------------------------------------------
19  // Linear Containers
20  // A. Fischer June 12, 2001                                  file: linear.hpp
21  // ------------------------------------------------------------------------
22  #ifndef LINEAR_H
23  #define LINEAR_H
24  #include "contain.hpp"
25  #include "cell.hpp"
26  #include "tools.hpp"
27
28  class Linear: public Container {
29  protected: // ------------------------------------------------------------
30      Cell* head;      // This is a dummy header for the list.
31
32  private:   // ------------------------------------------------------------
33      Cell* here;      // Cursor for traversing the container.
34      Cell* prior;     // Trailing pointer for traversing the container.
35
36  protected: // ------------------------------------------------------------
37      Linear(): head(new Cell), here( nullptr ), prior( head ) {}
38      virtual ~Linear ();
39      void  reset()           { prior = head; here = head->next; }
40      bool  end() const       { return here == nullptr; }
41      void  operator ++();
42
43      //virtual void  insert( Cell* cp );
44      void  insert( Cell* cp );
45      virtual void  focus() = 0;
46      Cell* remove();
47      void  setPrior(Cell* cp){ prior = cp; here = prior->next; }
48
49  public: // ---------------------------------------------------------------
50      void     put(Item * ep) { if (ep) insert( new Cell(ep) ); }
51      Item*    pop();
52      Item*    peek()         { focus(); return *here; }
53      //virtual ostream& print( ostream& out );
54      ostream& print( ostream& out );
55  };
56  inline ostream& operator<<(ostream& out, Linear& s) {return s.print(out); }
57  #endif
```

The two private data members in this class, together with the functions reset(), end(), and `++` permit us to start at the beginning of the container and visit each member sequentially until we get to the end. The reset() function sets the pointer named `here` to the first item in the container and sets `prior` to `nullptr`. The `++` operator sets `prior = here` and moves `here` to the next item. At all times, these two pointers will point to adjacent items in the container (or be nullptr). The end() function returns true if `here` has passed the last item in the container.

Three of the functions inherited from Container are no longer virtual, so the run-time dispatcher will not look in the Stack or Queue class for an overriding definition when put(), pop() or peek() is called from within Linear. One would expect put() to be virtual, since stacks and queues need different methods for putting an item into the container. However, put() simply wraps the item in a Cell, then delegates the actual insertion operation to insert(), which `is` virtual so that the specific and appropriate version of insert() in the derived class (Stack, Queue, etc.) will always be dispatched. We say that the Linear class `collaborates` with Stack or with Queue to handle the insertion task.

In a similar way, remove(), which is not virtual collaborates with focus(), which *is* virtual to focus the deletion on the proper element of the list. Since this is a pure virtual function, Linear is an abstract class that cannot be instantiated. The print() function is virtual for the same reason: to pass the responsibility to a derived class that is the experts on how printing should be done. The destructor is virtual because C++ requires a virtual destructor in classes that have virtual functions.

The Container class refers to Cells but does not define Cell. From that class, all we know is that Cell is

a helper class that must be used when information is placed into the Container. At that stage, a Cell could be the traditional linked-list cell or it could be just a typedef-synonym for Item*, appropriate for use with an array-based container. In the Linear class, we must commit to one representation or another, and we do commit to using a linked list. The definition given here for Cell is the usual two-part structure containing an Exam* and a Cell*. Nothing in Cell is virtual and everything is inline.

```
56  // ------------------------------------------------------------------------
57  // Linear Containers
58  // A. Fischer June 12, 2001                                file: linear.cpp
59  // ------------------------------------------------------------------------
60  #include "linear.hpp"
61  // ------------------------------------------------------------------------
62  Linear::~Linear () {
63      for (reset(); !end(); ++*this) {
64          delete prior->data;
65          delete prior;
66      }
67      delete prior->data;
68      delete prior;
69  }
70  // ------------------------------ Move index to next Item in the container.
71  void
72  Linear::operator ++() {
73      if (!end() ){
74          prior = here;
75          here = here->next;
76      }
77  }
78  // ---------------- Put an Item into the container between prior and here.
79  // ------------- Assumes that here and prior have been positioned already.
80  void
81  Linear::insert(Cell* cp) {
82      cp->next = here;
83      here = prior->next = cp;
84  }
85  // ---------------- Take an Item out of the container. Like pop or dequeue.
86  // -- Assumes that here and prior have been positioned to the desired Item.
87  Cell*
88  Linear::remove() {
89      if (! here ) return nullptr;
90      Cell* temp = here;                  // Grab cell to remove.
91      here = prior->next = temp->next;    // Link around it.
92      return temp;
93  }
94  // ----------------------------------- Remove a Cell and return its Item.
95  Item*
96  Linear::pop(){
97      focus();                // Set here and prior for deletion point.
98      Cell* temp = remove();  // Remove first real cell on list.
99      if (!temp) return nullptr;  // Check for empty condition.
100     Item* answer = *temp;   // Using cast coercion from Cell to Item.
101     delete temp;            // Take contents out of cell, delete Cell.
102     return answer;          // Return data Item.
103 }
104 // ---------------------------------------- Print the container's contents.
105 ostream&
106 Linear::print (ostream& out ) {
107     out << "  <[\n";
108     for (reset(); !end(); ++*this)  out << "\t" <<*here;
109     return out << "  ]>\n";
110 };
```

As in previous linked list definitions, this Cell class gives friendship to Linear. One slight difference is the friendship declaration on line 122, which is needed for the operator extension on line 139. We need to choose one of three alternatives:

a. Using the friend function declaration,

b. Making the print function public, and

c. Not having an operator extension for class Cell.

In previous versions of this class, we chose strategy (c); this time we choose (a) because it makes the Linear::print function shorter.

15.3.3 Cell: The Helper Class

```
111  // ------------------------------------------------------------------------
112  // A cell contains an Item* and a link.  Cells are used to build lists.
113  // A. Fischer   June 13, 2000                              file: cell.hpp
114  // ------------------------------------------------------------------------
115  #ifndef CELL_H
116  #define CELL_H
117  #include "item.hpp"
118  #include "tools.hpp"
119  // ------------------------------------------------------------------------
120  class Cell {
121    friend class Linear;
122    friend ostream& operator<<( ostream& out, Cell& c);
123
124    private:  // ---------------------------------------------------------
125      Item* data;
126      Cell* next;
127
128      Cell(Item* e = nullptr, Cell* p = nullptr ): data(e), next(p){ }
129      ~Cell(){ cerr <<"\n  Deleting Cell 0x" <<this << dec <<"..."; }
130      operator Item*() { return data; }   // Cast Cell to Item*. ------------
131
132      void print(ostream& out) const {    // --------------------------------
133          if (data) {
134              out << "Cell 0x" << this;
135              out << " [" << *data << ", " << next << "]\n";
136          }
137      }
138  };
139  inline ostream& operator<<(ostream& out, Cell& c){c.print(out); return out;}
140  #endif
```

One new technique is introduced in Cell: we define a cast (line 130) from type Cell to type Exam*. This cast can be used explicitly, just like a built-in cast function or it can be used by the compiler to *coerce* (automatically convert) the type of an argument. For example, Line 100, in Linear::pop(), could be written two ways, as shown below.

```
Item* answer = (Item*)(*temp);    // Explicit use of a cast from Cell to Item*.
Item* answer = *temp;              // Using cast coercion from Cell to Item*.
```

The first uses cast operation explicitly. The second supplies `*temp` in a context where an Item* is required. The compiler will find the cast operator on line 130 and use it to coerce the argument on the right to the type of the variable on the left.

15.3.4 Exam: The Actual Data Class

The linear containers defined above would be appropriate to store any kind of data. The data class used here is very simple but could be much more complex. It is used again in the next chapter where it is extended by

adding comparison functions and a key() function so that the data can be sorted.

```
141  //============================================================================
142  //  Exam: A student's initials and one exam score.
143  //  A. Fischer, October 1, 2000                                file: exam.hpp
144  //============================================================================
145  #ifndef EXAM_H
146  #define EXAM_H
147  #include "tools.hpp"
148
149  typedef int KeyType;
150  //----------------------------------------------------------------------------
151  class Exam                      // One name-score pair
152  {
153    private:  //-------------------------------------------------------------
154      char Initials [4];          // Array of char for student name
155    protected://-------------------------------------------------------------
156      int Score;                  // Integer to hold score
157    public:  //--------------------------------------------------------------
158      Exam (const char* init, int sc){
159          strncpy( Initials, init, 3 );
160          Initials[3] = '\0';
161          Score = sc;
162      }
163      ~Exam (){ cerr << "    Deleting Score " <<Initials <<"..."; }
164      ostream& Print ( ostream& os ){
165          return os <<Initials <<": " <<Score <<"  ";
166      }
167  };
168  //----------------------------------------------------------------------------
169  inline ostream& operator << (ostream& out, Exam& T){ return T.Print( out ); }
170  #endif

174  //============================================================================
175  //  Bind abstract name ITEM to the name of a real class.
176  //  A. Fischer, June 15, 2000                                  file: item.hpp
177  //============================================================================
178  #ifndef ITEM_H
179  #define ITEM_H
180  typedef Exam Item;
181  #endif
```

15.3.5 Class diagram.

Figure 15.3.5 is a UML diagram for this program, showing the polymorphic nature of Linear and its relation to Cell, Stack, and Queue.

15.4 Stack: Fully Specific

A Stack is a container that implements a LIFO discipline. In this program, we derive Stack from Linear and, in so doing, we represent a stack by a linear linked list of Cells. From Linear, Stack inherits a head pointer, two scanning pointers (here and prior) and a long list of functions.

Notes on the Stack code.

- The list head pointer is protected, not private in Linear, because some derived classes (for example Queue), need to refer to it. The scanning pointers are private because the derived classes are supposed to use Linear::setPrior() rather than setting the pointers directly. This guarantees that the two scanning pointers are always in the correct relationship to each other, so that insertions and deletions will work properly. It also saves lines of code in the long run, since they are written once in Linear instead of multiple times in the derived classes.

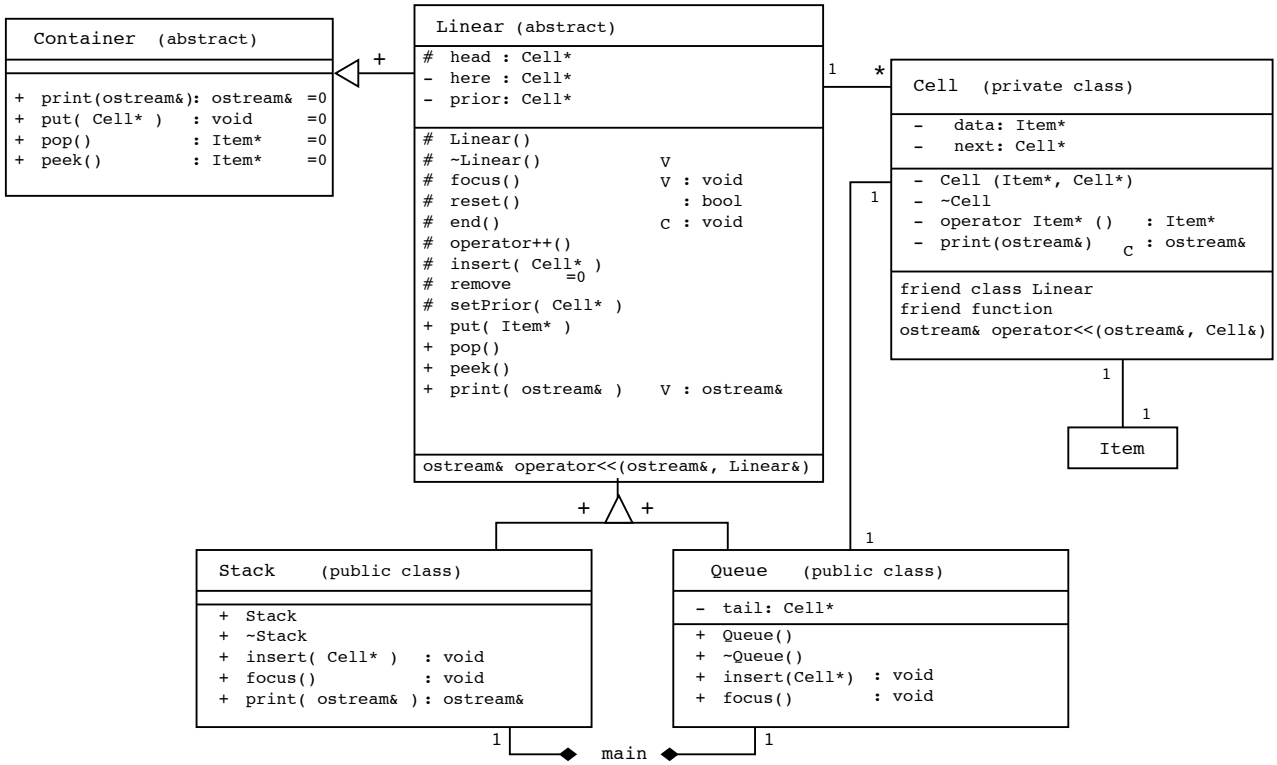

Figure 15.3: UML diagram for the Polymorphic Linear Container

- The Stack constructor and destructor are both null functions; they are supplied because it is bad style to rely on the defaults. The program does work perfectly well without either. Five functions are defined explicitly in Stack.

```
182  // ------------------------------------------------------------------------
183  // Stacks, with an inheritance hierarchy
184  // A. Fischer  June 8, 2001                                  file: stack.hpp
185  // ------------------------------------------------------------------------
186  #ifndef STACK_H
187  #define STACK_H
188  #include "linear.hpp"
189
190  // ------------------------------------------------------------------------
191  class Stack : public Linear {
192    public:
193      Stack(){}
194      ~Stack(){}
195      void  insert( Cell* cp ) { reset(); Linear::insert(cp); }
196      void  focus(){ reset(); }
197
198      ostream& print( ostream& out ){
199          out << "  The stack contains:\n";
200          return Linear::print( out );
201      }
202  };
203  #endif
```

- Stack defines focus(), which is abstract in Linear. This is called by Linear::remove() to set prior and here so that the first Cell in the container will be returned. (Last in, first out.)

- Since focus(), was the only remaining pure virtual function in Linear. Since we define it here, Stack becomes a fully specified class and can be used to create objects (that is, it can be instantiated.)

- Linear::put(Exam*) calls Insert, which is defined in both Linear and Stack. Because insert() is virtual, the version in Stack will be used to execute the call written in Linear::put(). We want to insert the new Cell at the head of the list because this is a stack. The actual insertion will be done by Linear::insert(), after Stack::insert() positions here and prior to the head of the list by calling reset(). The left side of figure 15.4 illustrates how control passes back and forth between the base class and the derived class during this process.

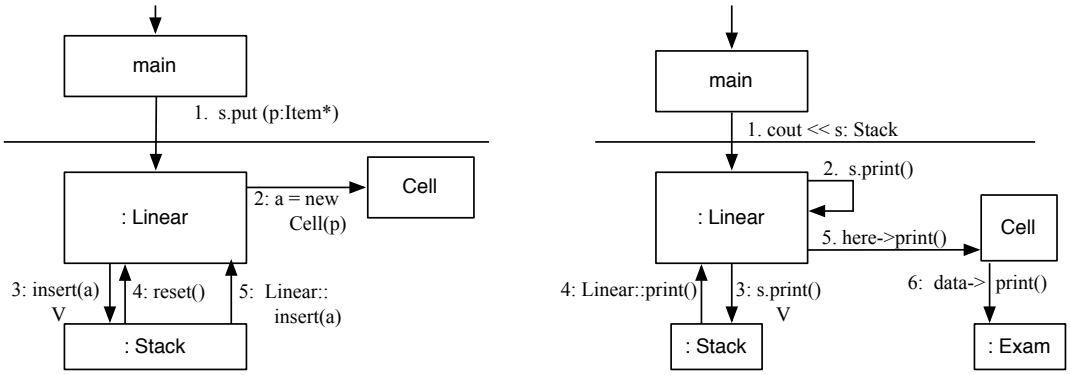

Figure 15.4: How the classes collaborate during Stack::put (left) and Stack::print (right).

- The redefinition of print() is not absolutely necessary here; all it does is to print a few words ("The stack contains:") and call Linear::print() to print the contents. For contrast, the Queue class does not define a print function. However, the collaboration between the two print() functions and the `<<` operator is instructive.

  In the main program, we never call the print functions directly. All printing is done by statements like `cerr << S;` But Stack and Queue do not supply definitions for `operator<<`, so the output task is given (by inheritance) to the definition in Linear (line 54). It, in turn, calls Linear::print(), which is virtual, and dispatches the job to Stack::print() because S is a stack. Stack::print() prints the output label and calls Linear::print() to finish the job. The class name is necessary only in the last call; all other shifts of responsibility are handled by inheritance or virtual dispatching. This activity is diagrammed on the right in Figure 15.4

15.5 Queue: Fully Specific

```
204  // ------------------------------------------------------------------------
205  // Queues: derived from  Container-<--Linear-<--Queue
206  // A. Fischer   June 9, 2001                                  file: queue.hpp
207  // ------------------------------------------------------------------------
208  #ifndef QUEUE_H
209  #define QUEUE_H
210  #include "linear.hpp"
211
212  // ------------------------------------------------------------------------
213  class Queue : public Linear {
214    private:
215      Cell*   tail;
216
217    public:           // ------------------------------------------------------
218      Queue() { tail = head; }
219      ~Queue(){}
220
221      void  insert( Cell* cp ) { setPrior(tail); Linear::insert(cp); tail=cp;}
222      void  focus(){ reset(); }
223  };
224  #endif
```

A queue is a container that implements a FIFO discipline. This Queue is a linear linked list of Cells with a dummy header. This declaration of Queue follows much the same pattern as Stack. However, Queue uses the inherited Linear::print(), instead of defining a specific version of its own. This is done for demonstration purposes; we recognize that better and more informative formatting could be achieved by defining a specific local version of print().

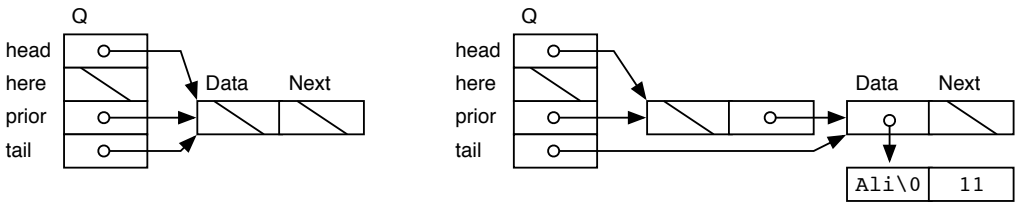

Figure 15.5: The Queue named Q, newly constructed and after one insertion.

Notes on the Queue code.

- Linear has three data members (head, prior and here), and Queue has one (tail). The constructor of each class must initialize the data members in its own class. When a Queue is constructed, all four are initialized to create an empty linked list with a dummy header, as in Figure 15.5. An important issue here is that the constructor of the base class, Linear, is executed first. It creates a dummy cell and attaches it to the head and prior pointers. When control gets to the Queue constructor, the dummy cell will exist and it is easy to attach the tail pointer to it.
- Queue::insert(Item*) set here and prior to the tail of the list so that insertions will be made after the last list cell. As before, Linear::insert() then finishes the job and does the actual insertion. Almost no code is duplicated because the classes collaborate.
- Queue defines focus(), which is abstract in Linear, to set the scanning pointers to the beginning of the Queue. The result is that the earliest insertion will be the next removal (FIFO).
- When an inherited virtual function (such as pop) is called, control passes back and forth between the base class and the derived class, as shown in Figure 15.4.

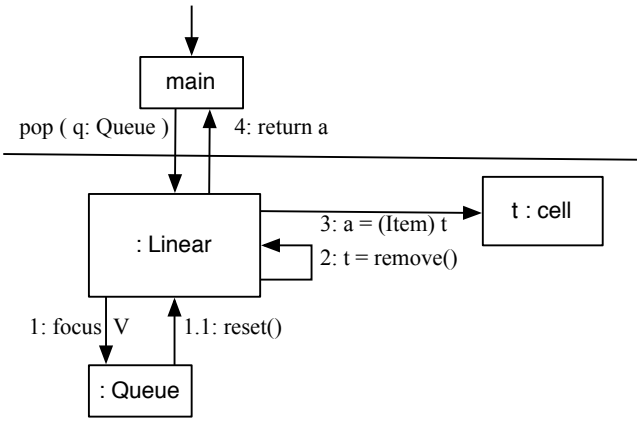

Figure 15.6: How the classes collaborate during Queue::pop().

- The output (following main) shows that the queue Q was printed by the inherited function, Linear::print().
- If Linear::insert() *is not* virtual, insertion is done incorrectly for queues. I removed the "virtual" property and got the following output :

```
Putting 3 items on the Queue Q: 11, 22, 33.
  ]>
  <[    Cell 0x0x804c498 [Cil: 33  , 0x804c478]
        Cell 0x0x804c478 [Bea: 22  , 0x804c458]
        Cell 0x0x804c458 [Ali: 11  , (nil)]
  ]>
```

Compare it to the correct output at the end of this chapter, where Ali comes first in the queue.

15.6 A Main Program and its Output

```
226  // ------------------------------------------------------------------------
227  // Demonstration of derived classes with virtual functions.
228  // A. Fischer  June 15, 1998                                file: main.cpp
229  // ------------------------------------------------------------------------
230  #include "tools.hpp"
231  #include "exam.hpp"     // Must precede #include for item.hpp.
232  #include "item.hpp"     // Abstract base type for stacks and queues.
233  #include "stack.hpp"    // Base type is Item == Exam.
234  #include "queue.hpp"    // Base type is Item == Exam.
235  // ------------------------------------------------------------------------
236  int main( void ) {
237      Stack S;
238      Queue Q;
239
240      cerr << "\nPutting 3 items on the Stack S: 99, 88, 77.\n" ;
241      S.put( new Exam("Ned", 99) );       //cerr << S << endl;
242      S.put( new Exam("Max", 88) );       //cerr << S << endl;
243      cerr << "  Peeking after second insertion: " <<*S.peek() <<"\n";
244      S.put( new Exam("Leo",77) );          cerr << S << endl;
245
246      cerr << "Putting 3 items on the Queue Q: 11, 22, 33.\n";
247      Q.put( new Exam("Ali",11) );        //cerr << Q << endl;
248      Q.put( new Exam("Bea",22) );        //cerr << Q << endl;
249      cerr << "  Peeking after second insertion: " <<*Q.peek() <<"\n";
250      Q.put( new Exam("Cil",33) );          cerr << Q << endl;
251
252      cerr << "Pop two Exams from Q, put on S. \n";
253      S.put(Q.pop()); S.put(Q.pop());       cerr <<"\n" <<S << endl;
254
255      cerr << "Put another Exam onto Q: 44.\n";
256      Q.put( new Exam("Dan",44) );          cerr << Q << endl;
257
258      cerr << "Pop two Exams from S and discard.\n";
259      delete S.pop();
260      delete S.pop();                       cerr <<"\n" << S << endl;
261      bye();
262  }
```

To test the linear container classes, we wrote a meaningless main program that instantiates one Stack and one Queue and moves data onto both and from one to the other. The call graph in Figure 15.6 attempts to show the ways that functions are actually called, after dynamic dispatching. In this chart, grey circles that say "V" mark virtual dispatching and circles that say "I" show calls that were made through inheritance.

Enough function calls are made to demonstrate that the classes work properly; the contents of S and Q are printed just often enough to see the data move into and out of each container. Note that several diagnostic output commands were used during debugging and are now commented out.

- The Queue implements a first-in first-out order.
- The Stack implements a last-in first-out order.
- peek() returns the same thing that pop() would return, but does not remove it from the list.
- Stack::print() is used to print the stack but the inherited Linear::print() prints the queue.
- All dynamically allocated objects are properly deleted.

The output:

```
Putting 3 items on the Stack S: 99, 88, 77.
  Peeking after second insertion: Max: 88
  The stack contains:
  <[    Cell 0x0x804c988 [Leo: 77  , 0x804c968]
        Cell 0x0x804c968 [Max: 88  , 0x804c948]
        Cell 0x0x804c948 [Ned: 99  , (nil)]
  ]>
```

```
Putting 3 items on the Queue Q: 11, 22, 33.
  Peeking after second insertion: Ali: 11
  <[    Cell 0x0x804c9a8 [Ali: 11  , 0x804c9c8]
        Cell 0x0x804c9c8 [Bea: 22  , 0x804c9e8]
        Cell 0x0x804c9e8 [Cil: 33  , (nil)]
  ]>

Pop two Exams from Q, put on S.

  Deleting Cell 0x0x804c9a8...
  Deleting Cell 0x0x804c9c8...
  The stack contains:
  <[    Cell 0x0x804c9c8 [Bea: 22  , 0x804c9a8]
        Cell 0x0x804c9a8 [Ali: 11  , 0x804c988]
        Cell 0x0x804c988 [Leo: 77  , 0x804c968]
        Cell 0x0x804c968 [Max: 88  , 0x804c948]
        Cell 0x0x804c948 [Ned: 99  , (nil)]
  ]>

Put another Exam onto Q: 44.
  <[    Cell 0x0x804c9e8 [Cil: 33  , 0x804ca08]
        Cell 0x0x804ca08 [Dan: 44  , (nil)]
  ]>

Pop two Exams from S and discard.

  Deleting Cell 0x0x804c9c8...    Deleting Score Bea...
  Deleting Cell 0x0x804c9a8...    Deleting Score Ali...
  The stack contains:
  <[    Cell 0x0x804c988 [Leo: 77  , 0x804c968]
        Cell 0x0x804c968 [Max: 88  , 0x804c948]
        Cell 0x0x804c948 [Ned: 99  , (nil)]
  ]>
```

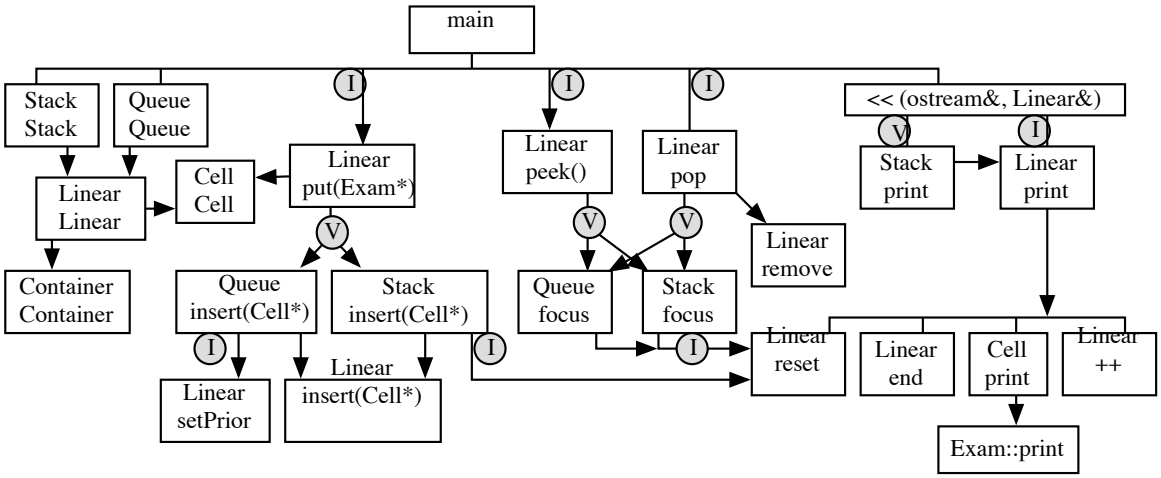

Figure 15.7: A call graph for the linear container program.

Termination. After printing the termination message, the objects Q and S will go out of scope and be deallocated. An output trace can serve as part of a proof that deallocation is done correctly and fully, without crashing. The output trace from main is given below, with a call graph (Figure 15.6) showing how control moves through the destructors of the various classes.

```
Normal termination.

  Deleting Cell 0x0x804c928...    Deleting Score Cil...
  Deleting Cell 0x0x804c9e8...    Deleting Score Dan...
  Deleting Cell 0x0x804ca08...
  Deleting Cell 0x0x804c918...    Deleting Score Leo...
  Deleting Cell 0x0x804c988...    Deleting Score Max...
  Deleting Cell 0x0x804c968...    Deleting Score Ned...
  Deleting Cell 0x0x804c948...
```

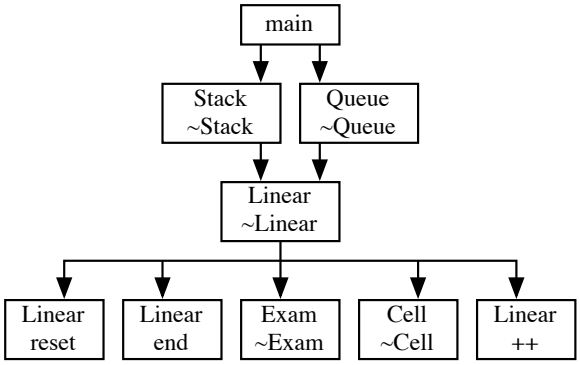

Figure 15.8: Terminating the linear container program.