Chapter 14: Derivation and Inheritance

In life and in understanding a complex program...

A picture is worth a thousand words.

This chapter consists of one interactive game program that uses templates, derivation, a makefile, a stringstore, a flexarray, and some C coding "tricks" that are worth knowing. The game output is presented first to familiarize you with how the game works. Following that are the makefile, main program, and pairs of .hpp and .cpp files, with notes on each. These are documented by several kinds of "pictures", each of which illustrates a different aspect of the application: a module dependency chart, a data structure diagram, UML class diagrams, and a function call chart.

14.1 Playing

This program plays an interactive word-guessing game called hangman. In this game, the leader (the computer) selects a secret word and displays a line of dashes with one dash for each letter. The player must guess letters, one at a time, and try to figure out what the hidden word is. A sample game is included here, for those who are unfamiliar with it. Suppose the computer chose "hippopotamus" as the secret word. The player would see:

```
---------Constructing Hangman ----------
Please enter name of vocabulary file (or ENTER to quit): vocab2.in

---------- Welcome to Hangman ----------
You win if you can guess the hidden word.
You lose if you guess 7 wrong letters.


Puzzle is: <[ _ _ _ _ _ _ _ _ _ _ _ _ ]>

    Letters left--> a b c d e f g h i j k l m n o p q r s t u v w x y z
    Bad guesses---> _ _ _ _ _ _ _
Guess a letter:
```

After one wrong guess (e) and one correct guess (a), the board would look like this:

```
 You guessed 'a'.  You scored!


Puzzle is: <[ _ _ _ _ _ _ _ _ a _ _ _ ]>

    Letters left--> b c d f g h i j k l m n o p q r s t u v w x y z
    Bad guesses---> e _ _ _ _ _ _
```

After several more guesses (i,o,y,u,t,p,m, and finally s), the game is won and you see:

```
 Puzzle is: <[ h i p p o p o t a m u s ]>

    Letters left--> b c d f g j k l n q r v w x z
    Bad guesses---> e y _ _ _ _ _
Congratulations -- you win!

You won 1 time out of 1 try.

Type p to play another round, q to quit: q

----------- Have a good day! -----------
```

14.1.1 The Hangman Application

A makefile defines its application: it lists the required parts and describes the relationships among them. The information in the makefile is presented graphically in Figure 14.1.

The makefile.

```
1   # Rule for building a .o file from a .cpp source file -------
2   .SUFFIXES: .cpp
3   .cpp.o:
4       c++ -c $(CXXFLAGS) $<
5
6   # Compile with debug option and all warnings on. ------------
7   CXXFLAGS = -g -Wall
8
9   # Object modules comprising this application ----------------
10  OBJ = main.o game.o board.o sstore.o rstrings.o words_d.o tools.o
11
12  game: $(OBJ)
13      c++  -o game $(CXXFLAGS) $(OBJ)
14
15  # Delete .o and exe files and force recompilation. ----------
16  clean:
17      rm -f $(OBJ) game
18
19  # Use tools source file from grandparent directory ----------
20  tools.o:   tools.cpp tools.hpp
21      c++ -c $(CXXFLAGS) tools.cpp -o tools.o
22
23  # Dependencies ----------------------------------------------
24  main.o:     main.cpp game.hpp  board.hpp flexT.hpp
25  game.o:     game.cpp game.hpp board.hpp rstrings.hpp sstore.hpp flexT.hpp
26  board.o:    board.cpp board.hpp words_d.hpp
27  words_d.o:  words_d.cpp words_d.hpp words.hpp tools.hpp
28  rstrings.o: rstrings.cpp rstrings.hpp sstore.hpp flexT.hpp
29  sstore.o:   sstore.cpp sstore.hpp tools.hpp
```

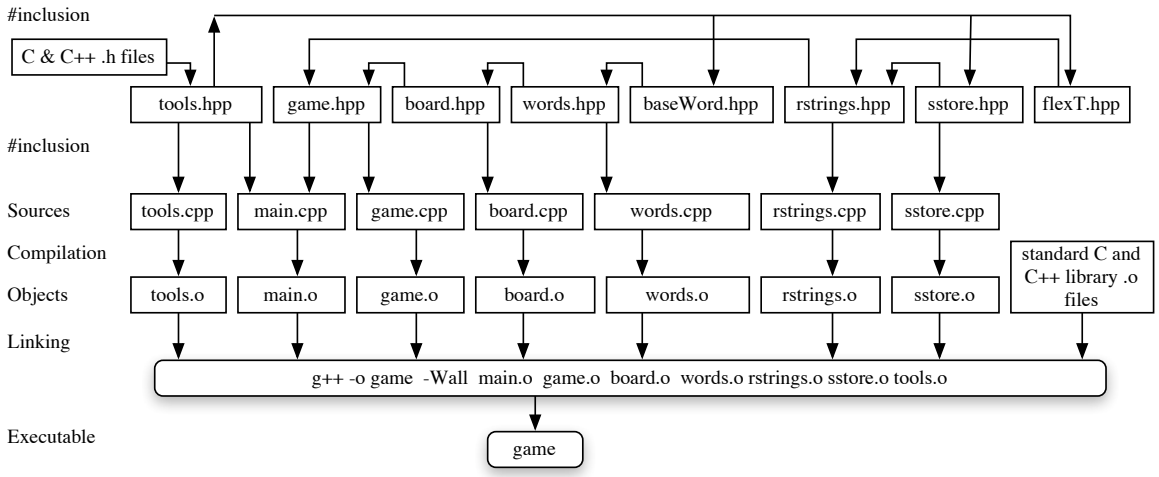

Figure 14.1: Makefile graph for Hangman: The files and their dependencies.

14.1.2 Hangman: The Main Program

The vocabulary file. This program is designed to construct a hangman game then play one or more rounds of it. Line 46 asks whether the user has given the name of a vocabulary file as a command-line argument. If so, that file name is sent to the Game constructor. If not, the NULL pointer signals the Game constructor to use a default file.

A conditional operator is used appropriately in the argument list for the Game constructor. It tests a condition and returns something (a pointer) in either case. C syntax requires that the same type of object must be returned by both clauses of a conditional operator. In this program, the ? asks whether the user typed more than one thing (the program name) on the command line. If so, the additional command field is returned (it

should be a file name). If not, a NULL is returned. The overall code is simplified considerably by using the conditional operator instead of an if-else statement.

```
30  // ========================================================================
31  // Hangman program:    Let the user guess words from the vocabulary file.
32  // A. Fischer, May 13, 2001                                  file: main.cpp
33  #include "tools.hpp"
34  #include "game.hpp"
35  //------------------------------------------------------------------------
36  int main( int argc, char* argv[] )
37  {
38      char response;              // For query, "Play again?"
39      int wins = 0, rounds = 0;   // For keeping score.
40      const char* timeword;       // For grammatical output: time, times.
41      const char* tryword;        // For grammatical output: try, tries.
42
43      cout << "\n--------- Constructing Hangman ----------\n";
44      Game g( argc>1 ? argv[1] : NULL );      // Get optional file name.
45
46      cout << "\n---------- Welcome to Hangman ----------\n"
47              "You win if you can guess the hidden word.\n"
48              "You lose if you guess " << HANG_MAX << " wrong letters.\n";
49      do {
50          wins += g.play();                   // Play one round of game.
51          rounds++;
52          timeword = (wins == 1) ? "time" : "times";
53          tryword = (rounds == 1) ? "try" : "tries";
54          cout << "\nYou won " << wins << " " << timeword
55               << " out of " << rounds << " " << tryword
56               << ".\n\nType p to play another round, q to quit: ";
57          cin >> response;
58      } while (tolower(response) == 'p');
59      cout << "\n---------- Have a good day! ----------\n\n";
60  }
```

Instructions. The instructions given on lines 46 through 48 will not be repeated before each round. The loop on lines 49 through 58 plays one round and queries the user about whether to continue.

One round of the game. Line 50 plays one round of the game and returns the result: 1 for a win, 0 for a loss. The wins and losses are tallied and displayed when user asks to quit. Lines 52 and 53 use string variables and conditional operators to select singular or plural wording so that the final score message will be displayed in correct English. This kind of care is not necessary in student projects but makes a difference in the perceived quality of a commercial job.

14.1.3 Call Graphs

Different kinds of documentation lead to different insights into the structure of an application. An object diagram shows us the way our actual storage is organized and used at run time. A class diagram tells us what properties each kind of object has, and which ones can be used in other classes. A flow chart shows us possible execution paths. An event trace (also called a *sequence diagram* is like a flow chart but also shows how control passes back and forth between classes during execution. A fifth kind of graphical documentation is the call graph—a static chart showing which functions can call which other functions at some point in the program. Call graphs can be useful during debugging for tracking down all possible ways for control to get to any particular function.

Figure 14.1.3 is a call graph for Hangman. To minimize the complexity of the diagram and focus on the class functions, calls on fatal, new, delete, and iostream functions have been omitted from the chart. From this chart, you can see that execution is divided into two major phases. First, a game is constructed, then played. When one round is played, a random string is selected, a board is constructed, and finally, the board is played.

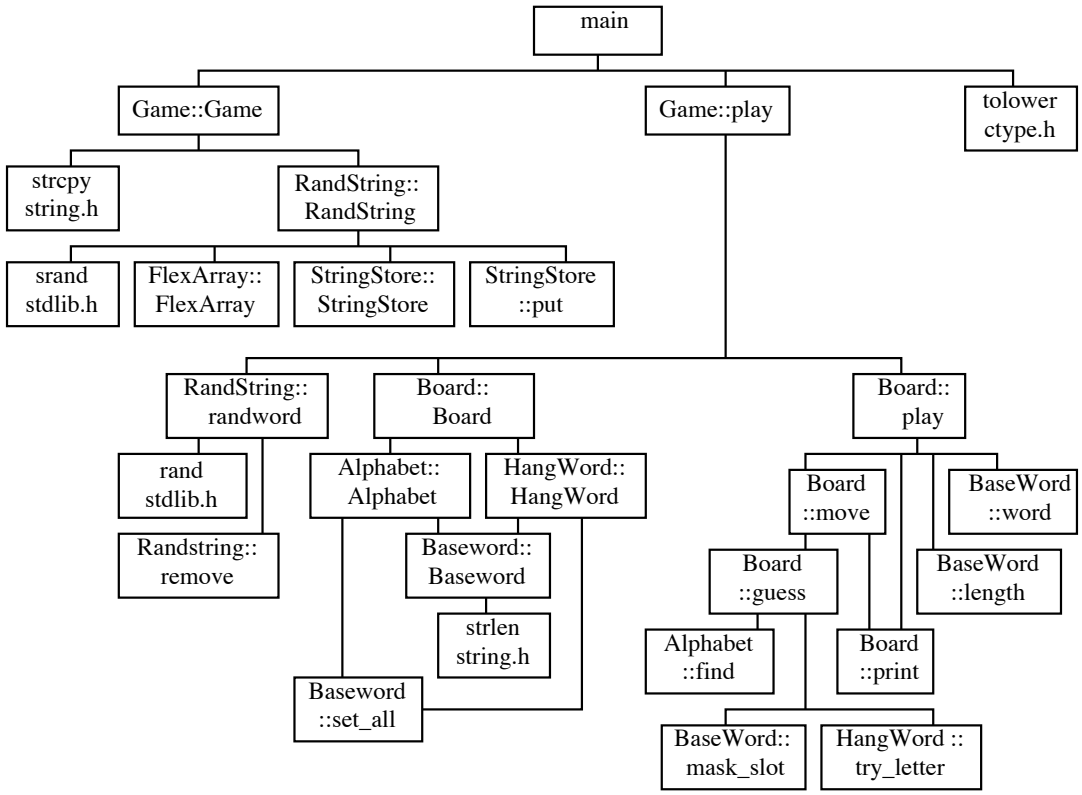

Figure 14.2: A call graph for the Hangman game: How control can reach each function.

14.1.4 UML Diagrams: A View of the Class Relationships

A UML diagram gives another static view of the application; it illustrates the data types being used, the protection level of each part of each class, and the possible ways that one class can access or use another.

The classes used in this application fall into two nearly separate subsystems: Game and Board, together with the three Word classes implement the form and function of the game itself, while RandString, StringStore, and FlexArray implement the database from which the game selects puzzle words. These two subsystems are diagrammed separately – the association between Game and RandString is the only connection.

In Figure 14.1.4 we see that two classes are derived from BaseWord. BaseWord defines a data structure and a set of functions that implement one basic behavior. From it, we derive two sub-classes that define variations on the basic theme and have different initialization, search, and display rules.

From the UML, you can see that the BaseWord class is not associated with or aggregated by any other classes and, in fact, is not instantiated by the program. It is used only as a basis for deriving Alphabet and Hangword, which form the basis for the gameboard display. This follows a basic OO-design guideline:

> Don't instantiate a class that you also derive from.

The goal of this rule is to minimize the conflict between the requirements of a base class, which must be clean and general, and a class that must serve the specific needs of an an application.

In Figure 14.1.4 we see an application-specific class, RandString, that is built out of two utility data-structure classes: it is derived from FlexArray and it aggregates StringStore. The RandString class is a container for words that randomly selects and returns one word at a time. The word is then removed from the container so it cannot be reused. In one sense, the new class, RandString "wraps" the two familiar classes in a new behavior, with only a fraction of the work that would be required to program the new class from scratch. It is a typical demonstration of the potential power of class libraries.

The FlexArray class has been rewritten, finally, in its proper form: as a class template. In going from FlexArray to RandString, we simultaneously bind the template parameter to `char*` and derive from the resulting class. This combination of template instantiation and derivation is very common because you normally want to write application-specific functions to handle the general data structures for which templates are used.

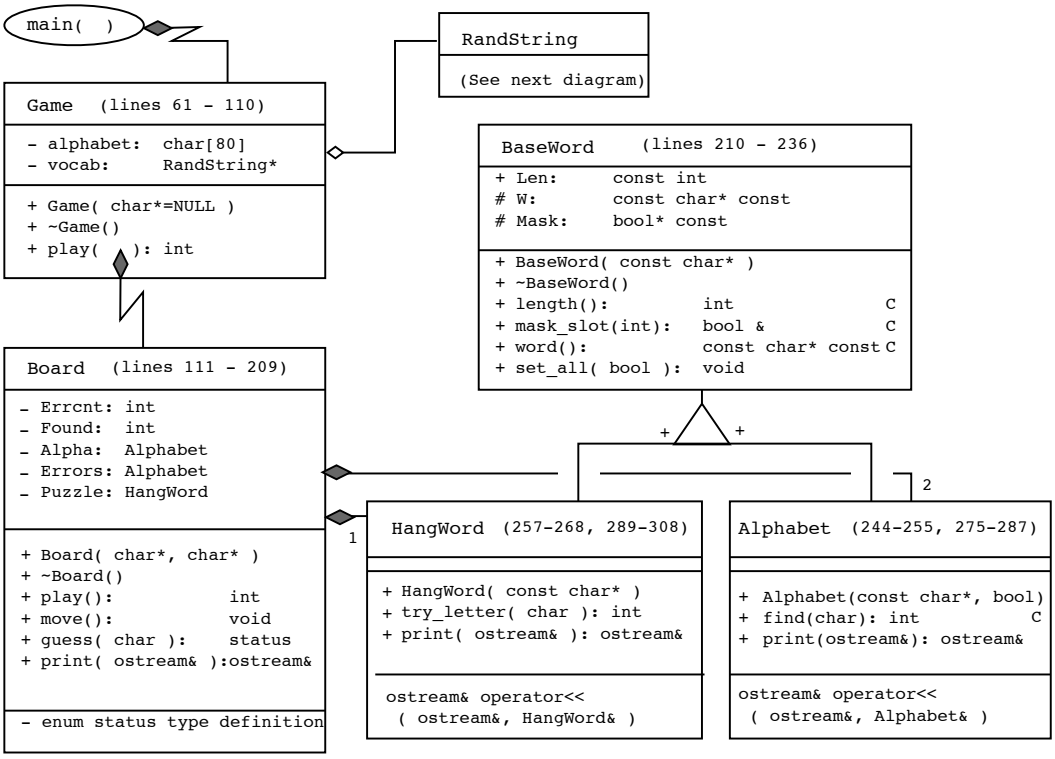

Figure 14.3: The game classes in Hangman: UML describes class relationships.

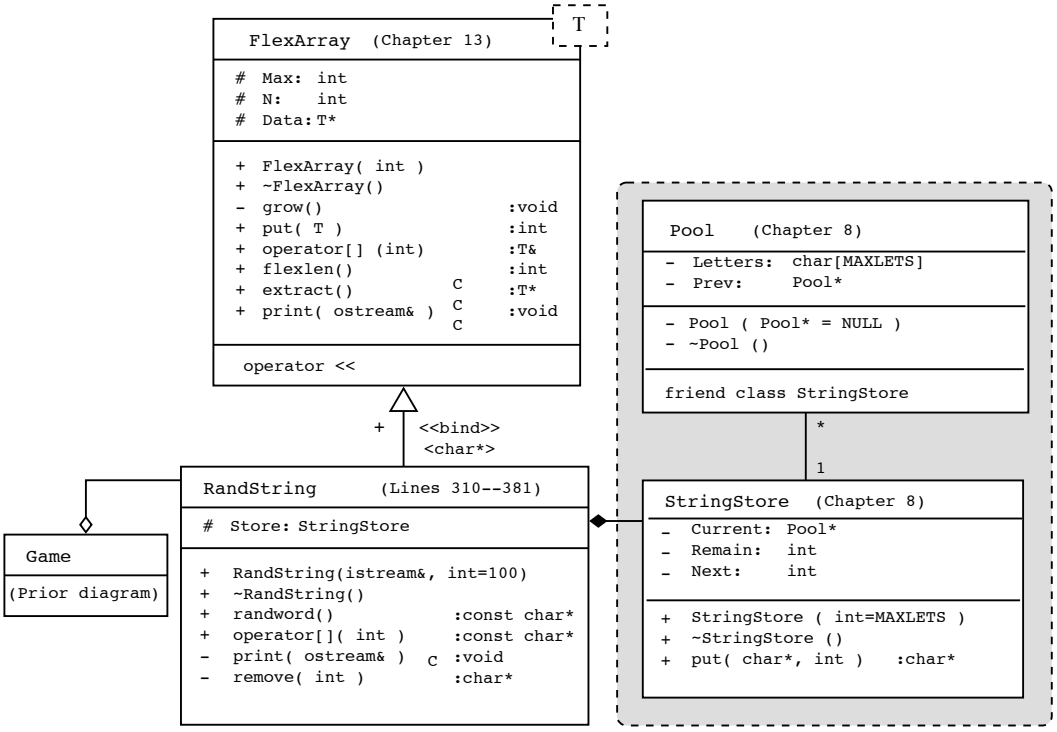

Figure 14.4: The vocabulary classes in Hangman.

14.1.5 The Hangman Data Structures

A data diagram illustrates the allocations, connections and contents that have been created by the program at one specific moment at run time, given a specified sequence of inputs. This kind of picture is particularly helpful when pointers are used to build a complex structure. It does not help us know how the program reached the particular state that is illustrated, but it gives us an appreciation for the overall complexity of the application. A data diagram can be a great help to a reader who is trying to understand how and why a program works (or fails to work).

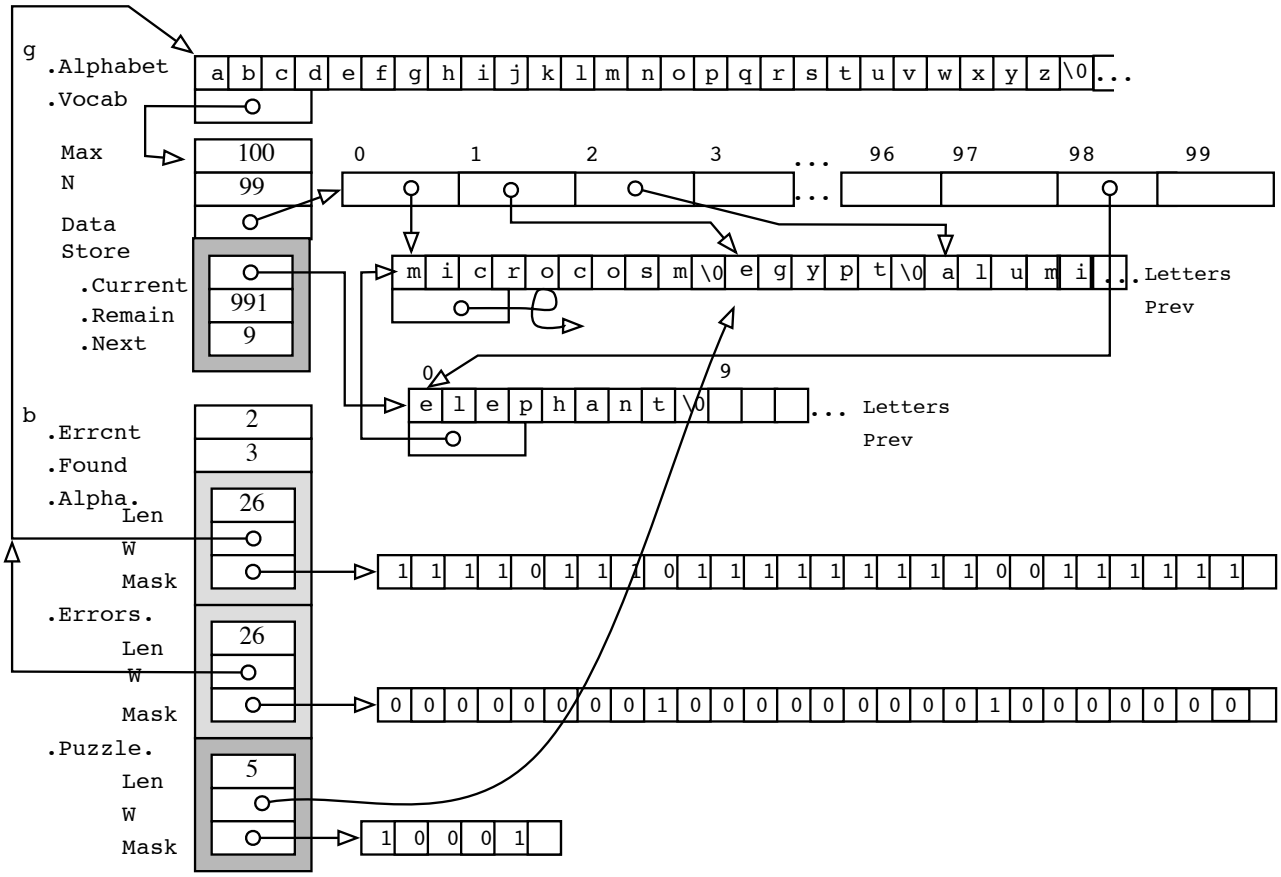

Figure 14.5: Data structures for the Hangman program: A snapshot during execution.

Program notes throughout this section will refer to the data diagram in Figure 14.1.5. It illustrates the data structures constructed for a representative game using a vocabulary file with 99 words, when the puzzle word is "egypt". The data structure, above, are shown after the fourth guess. (Two guesses were correct, two were wrong). Here is the current gameboard:

```
Puzzle is: <[ e _ _ _ t   ]>

    Letters left--> a b c d f g h j k l m n o p q r u v w x y z
    Bad guesses---> i s _ _ _ _ _
Guess a letter:
```

14.2 Hangman: The Game and Board Classes

The main program creates a Game (s 44) which, in turn, creates a Board. Game::play() calls RandString::randword() to select a random string from the vocabulary, builds a gameboard around it and plays one round of hangman. The Board class implements the gameboard with the help of Alphabet and HangWord, both derived from BaseWord.

A new gameboard is created by Game::play() (lines 50, 108) for each round. Since the array members of Board are built around the hidden word, each new word requires a new construction job. We could have done

the same thing by declaring a Board* as a member of Game, but there would be no advantage in doing it that way because there is no need to use a board after returning from Game::play().

14.2.1 The Game Class

```
61  // ========================================================================
62  // Build a board, play it, and return the score to main.
63  // A. Fischer, June 4, 2000                                  file: game.hpp
64  #ifndef GAME
65  #define GAME
66
67  #include "rstrings.hpp"
68  #include "board.hpp"
69  // ========================================================================
70  class Game {
71    private:
72      char Alphabet[80];      // Normally the English Alphabet.
73      RandString* Vocab;      // The randomized vocabulary list.
74
75    public:
76      Game( char*  wordfile = NULL );
77      ~Game(){ delete Vocab; }
78      int play();             // Play hangman.
79  };
80  #endif

81  // ========================================================================
82  // Implementation of the Hangman game.
83  // A. Fischer, June 4, 2000                                 file: game.cpp
84
85  #include "game.hpp"
86  // ========================================================================
87  Game::Game( char* wordfile ){
88      char file_name[80];
89      //cerr << "Constructing Game.  ";       // For debugging.
90      if (wordfile) strcpy( file_name, wordfile );
91      else strcpy( file_name, "vocab.in" );
92
93      ifstream source( file_name );
94      if (!source) fatal( "Could not open %s.\nEnding Hangman.\n", file_name );
95      //cerr << "File " <<file_name <<" is open for reading." <<endl;
96
97      source.getline( Alphabet, 80 );
98      Vocab = new RandString( source );       // Create vocabulary database.
99      source.close();
100     //Vocab->print( cerr );                 // For debugging.
101 }
102
103 //-------------------------------------------------------------------------
104 int Game::play() {
105     const char* puzzle_word = Vocab->randword();    // pick puzzle word.
106     //cerr << puzzle_word;
107
108     Board b( Alphabet, puzzle_word );               // construct game board
109     return b.play();                                // play the round
110 }
```

Notes on the Game code. The Game constructor's first task is to find and open the vocabulary file. If no name was typed on the command line, wordfile will be NULL, otherwise it will point to the selected filename. A filename must be supplied, one way or another. If it was not given on the command line, the program uses a default file, `vocab.in`. In either case, a copy of the file name is copied into the local array and used to open an input stream.

Once the stream is open, the first line of the file is read (line 97) and used to initialize the class member named `Alphabet`. This permits the program to be used with vocabulary from any language. The remainder of the file is used (line 98) to build the vocabulary array (a new RandString structure). the result is shown at the top of Figure 14.1.5: g.Vocab points to a RandString object consisting of a FlexArray and a StringStore with two Pools.

The Game::play() function is the heart of the program: it selects a mystery word (line 105), uses it to construct a board (line 108), plays the board (line 109), and returns the score (1 win or 0 wins) to main(). Details of how to play the board and how to calculate the score are delegated to the Board class.

14.2.2 The Board Class

```
111  // ============================================================================
112  // Choose a word, use it to create a playing board.
113  // A. Fischer, June 4, 2000                                  file: board.hpp
114  #ifndef BOARD
115  #define BOARD
116
117  #include "words_d.hpp"
118  #define HANG_MAX 7
119  // ============================================================================
120  class Board {
121      enum status {GOOD_GUESS, BAD_GUESS, NOT_IN_ALPHA, USED_ALREADY};
122      int Errcnt;         // Wrong guesses so far,
123      int Found;          // Number of Letters correctly filled in.
124      Alphabet Alpha;     // Masked alphabet.
125      Alphabet Errors;    // Masked alphabet for error list.
126      HangWord Puzzle;    // Masked mystery word.
127
128    public:
129      Board(const char* a, const char* puz);  // Alphabet and puzzle word.
130      ~Board(){}
131      int play();                             // play a board
132      void move();                            // user interaction for one move
133      status guess(char c);                   // process a guess
134      ostream& print(ostream&);               // print a board
135  };
136  #endif


137  // ============================================================================
138  // board.cpp: Implementation for hangman board
139  // A. Fischer, June 4, 2000                                  file: board.cpp
140
141  #include "board.hpp"
142  // ============================================================================
143  Board::Board(const char* a, const char* puz) :
144      Errcnt(0), Found(0), Alpha(a, true), Errors(a, false), Puzzle(puz){
145      //cerr << "\nConstructing Board.  ";
146  }
147
148  //------------------------------------------------------- display the board
149  ostream&
150  Board::print( ostream& out ) {
151      out << "\n\nPuzzle is: " << Puzzle << "\n\n";
152      out << "    Letters left-->" << Alpha << "\n";
153      out << "    Bad guesses--->" << Errors;
154
155      for (int k = Errcnt; k < HANG_MAX; k++) out << " _";
156      return out << endl;
157  }
158
159  //------------------------------------------------------- process a guess
```

```
160  Board::status
161  Board::guess(char c) {
162      int where = Alpha.find(c);
163      if (where == -1) return NOT_IN_ALPHA;
164      if ( !Alpha.mask_slot(where) ) return USED_ALREADY;
165      Alpha.mask_slot(where) = false;
166
167      int matches = Puzzle.try_letter(c);
168      if (matches <= 0) {
169          Errors.mask_slot(where) = true;
170          Errcnt++;
171          return BAD_GUESS;
172      }
173      Found += matches;
174      return GOOD_GUESS;
175  }
176
177  //------------------------------------------- user interaction for one move
178  void
179  Board::move() {
180      char ch;
181      cout << "Guess a letter: ";
182      cin >> ch;
183      cout << "You guessed '" << ch << "'";
184      switch (guess( ch )) {
185         case NOT_IN_ALPHA:
186              cout << " -- but it's not in the alphabet." << endl;    break;
187         case USED_ALREADY:
188              cout << " -- but you guessed it once before." << endl;  break;
189         case BAD_GUESS:
190              cout << " -- too bad." << endl;                         break;
191         case GOOD_GUESS:
192              cout << ".  You scored!" << endl;                       break;
193      }
194      print(cout);
195  }
196
197  //------------------------------------------------------------- play a board
198  int
199  Board::play() {
200      print(cout);
201      while (Errcnt < HANG_MAX && Found < Puzzle.length()) move();
202      if (Found == Puzzle.length()) {
203          cout << "Congratulations -- you win!" << endl;
204          return 1;
205      }
206      cout << "Sorry, you lose!" << "\nThe answer is: "
207           << Puzzle.word() << endl;
208      return 0;
209  }
```

To play a round of hangman, we need a hidden word and an alphabet. To make a good interface, we also need a list of wrong guesses. These three arrays of characters (Puzzle, Alpha, and Errors) are updated and displayed after every guess to help the player make skillful guesses. Thus, our gameboard displays three arrays of letters:

- Puzzle, the mystery-word consisting of underscores and correctly guessed letters.
- Errors, a list of incorrect guesses.
- Alpha, the list of letters that have not yet been guessed.

In addition, the Board object must keep score and end the round when the number of correct guesses (Found) equals the length of the puzzle word, or when the number of bad guesses (Errcnt) reaches the limit, `HANG_MAX`. In Figure 14.1.5, you see the Board, b, and its five components in the lower part of the diagram.

It is much easier to believe in the correctness of a function when its return values have meaningful names than when integer codes are used. For this reason, we define a private enumerated type to describe the possible outcomes of a guess. A value of this type is returned by guess() and used by move() to select a response message for the player. We use this device to clarify, simplify, and modularize the code. The resulting two functions are much clearer that the alternatives: one very long function and/or cryptic integer codes for the outcomes.

Constructing a playing board. The parameters to the constructor are an alphabet and a word that was randomly selected from the vocabulary. This code uses ctors (line 144) to initialize all of the data members, although they are only necessary for the last three. The body of the constructor contains only a debugging message, currently commented out. To return to a debugging phase, one would remove the // marks.

The last three ctors convey arguments from the parameter list of the Board constructor to the constructors of the aggregated class objects. The mystery word is sent to the Puzzle constructor where it becomes the basis for a new playing board. Class members Alpha and Errors are both constructed from the alphabet. Initially, all letters of the Alpha alphabet are set to "true", meaning that they are available, and all letters in the Errors alphabet are set to false, meaning that no errors have yet been made. These initializations will be discussed more fully in the relevant classes.

One round of play. Board::play() is called by Game::play() to play one round of hangman on a newly constructed Board. This function calls Board::move() in a loop (line 201), until the round has been won (all letters have been guessed, lines 202–204) or lost (seven errors have been made, lines 206–207). In both cases, the function announces the result to the player and returns the score to Game::play(). The details of how a move is made are handled by the move() function.

One move. Board::move() prompts for and reads a guess from the user, (line 181–183) calls Board::guess() to analyze the guess (line 184), and displays a message about the result and the new board position. The switch statement illustrates how well-chosen enumeration constants can make code much clearer. A reasonable question is, "why don't we combine the code from Board::move() and Board::guess() into one function and get rid of both the switch and the enumeration. There are two answers. First, the combined function would be very long. Second, it is very helpful to keep high-level logic separate from low-level logic. The switch implements the high-level logic and provides a road map for the other function. In contrast, Board::guess() is filled with many detailed comparisons and counters. The enumeration constants guide the reader and help clarify the purpose of the operations.

Is the guess correct? Given a puzzle, an alphabet, and a guess, there are four possible outcomes. We do a case analysis and use four return statements to simplify the logic by keeping the cases maximally separate from each other.

If the guess is not in the game-alphabet or it has been guessed previously, the player will not be "charged" for the bad guess. To find out, we call Alphabet::find() (line 162) to search for the guess among the legal letters. The return value is the subscript of the guess in the alphabet, if it is a legal letter, otherwise -1. For example, when 'e' is guessed, the subscript 4 will be returned because 'e' is the fifth letter in the alphabet.

If the letter is valid, we then check (line 164) whether it was previously guessed. To do this, we use the subscript returned by Alphabet::find() to index the mask array that parallels the alphabet array. (A result of false means the letter has been used, true means it is still available.)

If the letter is legal and still possible, we set the appropriate position in the mask array (line 165) to "false" to indicate that the letter is now used. Since 'e' has been guessed, we see that Alpha.Mask[4] is false and 'e' has disappeared from the display.

Next, on line 167, we determine whether the guess is correct or wrong by calling Hangword::try_letter() . The result will be the number of letters in the puzzle that match the guess (zero or more). If the answer is zero, we set the corresponding position of the error array to "true", increment the error-counter, and return with an error code (lines 168–171). This will cause the letter to "appear" (in alphabetical order) in the error array the next time it is displayed. In Figure 14.1.5, the letter 's' has been guessed and it is wrong. In response, Alpha.Mask[18] was set to false and Errors.Mask[18] to true. If the guess is correct, we reach line 173, where we

add the number of new matches to the match-counter (Found) and return a success code. Play for this board will end when the Found total equals the length of the mystery word.

The return type must be given here as "Board::status", even though it is written simply as "status" on line 121. This is necessary because the status type is defined inside the Board class and this function definition is outside the class declaration.

14.3 The Word Classes

A mask marks a subset. A masked data structure is the simplest way to denote a subset of the data that is to be used (or not used) for a particular purpose. One or several masks might be used to mark one or several different subsets. The mask field or fields might be members of the structure or might exist in a parallel data structure. The mask fields might be type bool (to denote a simple yes/no choice) or any other enumerated type. Masks are most useful if the condition that they represent is not simple to test for, and if the status of an object changes over time or is checked frequently.

A masked structure lets us sort or process data efficiently and easily. For example, suppose a club membership database has one record for each member. At least two sets of masks might be helpful: one for age (juvenile, teenager, adult, senior) and another for dues status (guest, lifetime, paid-up, due, lapsed). Masks can be useful here because age is messy to categorize and dues status is based on the member's history.

14.3.1 The BaseWord Declaration

The class BaseWord creates a basic masked string, that is, a string with a corresponding array of bools to indicate whether each letter in the string is currently valid or not. If a letter is valid, it is searched and displayed. If not, it is hidden both from sight and from the computations. You could say that the mask controls access to the individual letters in the array. This is an easy and fast way to select a subset of the data stored in an array. To add an element or remove it, simply change the bit from true to false, or vice versa.

```
210  // ============================================================================
211  // Maskable word base class.
212  // A. Fischer, June 4, 2000                                   file: words.hpp
213  #ifndef WORDS
214  #define WORDS
215  #include "tools.hpp"
216  // ============================================================================
217  class BaseWord {
218    protected:
219      const int Len;      // length of word, excluding terminator.
220      const char* const W;    // partially concealed word
221      bool* const Mask;       // concealment mask
222
223    public:
224      BaseWord(const char* st) : Len(strlen(st)), W(st), Mask(new bool[Len+1]){
225          //cerr << "\nConstructing BaseWord.  ";
226      }
227      ~BaseWord() { delete [] Mask; }
228      int length() const { return Len; }
229      bool& mask_slot(int k) const { return Mask[k]; }
230      const char* const word() const { return W; }
231
232      void set_all(bool on_off){  // set false to hide letter, true to expose.
233          for (int k=0; k<Len; k++) Mask[k] = on_off;
234      }
235  };
236  #endif
```

The BaseWord class. In this class, everything is based on a particular string (the argument to the BaseWord constructor) and its length. The members named Len and Mask are constants, so they must be initialized using ctors, *after* the length of the argument string is known. The first ctor that must be executed is the one that measures the string and stores its length in Len; the mask cannot be constructed until that is done. For this reason, we declare Len first in the class.

The class member named W is a constant pointer to a constant string, either the alphabet or a word that was selected randomly from the vocabulary and is still stored there. No copy is made of this string because we do not need to modify it. The word, itself, is constant. The mask array is modified during the play, and those modifications determine which letters are displayed.

14.3.2 The Derived Word Classes

```
237  // ============================================================================
238  // Derived Word classes.
239  // A. Fischer, June 4, 2000                                  file: words_d.hpp
240  #ifndef WORDSDERIVED
241  #define WORDSDERIVED
242
243  #include "words.hpp"
244  // ============================================================================
245  class Alphabet : public BaseWord {
246    public:
247      Alphabet(const char* st, bool on_off) : BaseWord(st) {
248          set_all(on_off);
249          //cerr << "Constructing Alphabet.";
250      }
251      int find(char c) const;     // return index of first c in word
252      ostream& print (ostream&);  // print an alphabet
253  };
254
255  inline ostream& operator<<(ostream& out, Alphabet& x){ return x.print(out); }
256
257  // ============================================================================
258  class HangWord : public BaseWord {
259    public:
260      HangWord(const char* st) : BaseWord(st) {
261          set_all(false);
262          //cerr << "Constructing HangWord. " <<st;
263      }
264      int try_letter(char);
265      ostream& print (ostream&);  // print a hang word
266  };
267
268  inline ostream& operator<<(ostream& out, HangWord& x){ return x.print(out); }
269  #endif

270  // ============================================================================
271  // Implementation for maskable words.
272  // A. Fischer, June 4, 2000                                  file: words_d.cpp
273
274  #include "tools.hpp"
275  #include "words_d.hpp"
276  // ================================================ Alphabet class functions
277  int                              // Return index of first occurrence of c in W
278  Alphabet::find(char c) const {
279      int k;
280      for (k=0; k<Len && c != W[k]; k++);     // Loop body is empty.
281      return (k==Len) ? -1 : k;
282  }
283  //----------------------------------------------------------------------------
284  ostream&
285  Alphabet::print (ostream& out) {
286      for (int k=0; k < Len; k++)  if ( Mask[k] ) out << " " << W[k];
287      return out;
288  }
289
290  // ================================================ Hangword class functions
291  int       // Count the number of times letter c occurs in puzzle; unmask each
```

```
292  HangWord::try_letter(char c) {
293      int count = 0;
294      for (int k=0; k<Len; k++) {
295          if (c == W[k]) {
296              count++;
297              Mask[k] = true;
298          }
299      }
300      return count;
301  }
302  //-------------------------------------------------------------------------
303  ostream&
304  HangWord::print (ostream& out) {
305      out << "<[" ;
306      for (int k=0; k < Len; k++) out  << ' ' << (Mask[k] ? W[k] : '_') ;
307      out << " ]>";
308      return out;
309  }
```

The Alphabet and HangWord classes. Baseword is a polymorphic class that defines a basic data structure and some of its functions. Alphabet and HangWord are variations of BaseWord. They are derived from BaseWord, so they inherit functions and data from BaseWord. In addition, each has its own set of functions for initialization, use, and display. The first line of a class declaration declares the derivation relationships, if any. In Hangman, Alphabet and HangWord are both derived from BaseWord by public derivation:

```
class Alphabet :  public BaseWord { ...  };
class HangWord :  public BaseWord { ...  };
```

From this relationship we know that:

1. The two new classes are derived from BaseWord so that we can have the same structure but different print functions.
2. The first part of an Alphabet or HangWord object is a BaseWord object.
3. The constructors for Alphabet and HangWord use ctor initializers to provide arguments for the BaseWord constructor.
4. The public/protected/private status of all the inherited members in Alphabet and HangWord is the same as in class BaseWord.
5. The functions of the two derived classes can freely use the protected members of the base class: Len, W, and Mask.

Initializing the masks. A gameboard contains three Words (Puzzle, Alpha, and Errors); each consisting of a char array with a parallel mask array. When a word is displayed, its mask array is checked; a letter in the word is displayed if its mask bit is on (true), and ignored if the bit is off.

During Board construction, the function BaseWord::set_all() is called to initialize the three Word members of the Board. It will set all of the bits of a mask to either true or false, depending on how the instance will be used: bits for the game alphabet are set to true, for the errors to false, and for the mystery word to false. The bool parameter for set_all() comes from a call in the HangWord constructor (line 261) or from the ctor for Alphabet in the Board constructor (line 144). As the game progresses, two mask bits are changed each time a legal guess (correct or incorrect) is made. The alphabet that is displayed grows shorter (as letters are used) and the error list grows longer (as letters are added to it). The mystery word stays the same length, but dashes in the display are replaced by letters each time a correct guess is made.

Printing through a mask. The polymorphic class lets us create different means of displaying the same data structure.When a word is displayed, its mask array is checked, and each letter in the word is displayed if its mask bit is on (true). For Alphabet objects, nothing is displayed if the bit is false, but for the puzzle (a HangWord), dashes are shown.

Searching and updating the masks. When the player guesses a letter, Board::guess() searches the alphabet (line 162) to find out whether the letter is legal and saves the index of the letter for later use. If the letter is not in the alphabet or if it has already been used, the function returns immediately with an error code (lines 163–164). Otherwise, it turns off the mask bit (line 165) corresponding to the letter to indicate that the letter has been used. (This removes the letter from the display.)

Then try_letter() is called (line 167) to compare the guessed letter to the letters in the puzzle word (lines 294–299). This function turns on the mask bit corresponding to each matching letter in Puzzle (line 297) and returns the number of matches (0 or more) that were found. If no matches were found, 0 is returned and the mask field in Error that corresponds to the bad guess is turned on (line 169). This causes the letter to appear on the error list. Finally, lines 173–174 add the number of matches to the score and return a success code. In Figure 14.1.5, Puzzle.Mask[0] and Puzzle.Mask[4] have been set to true in response to the two correct guesses, 'e' and 't'. The same two letters have been marked as false in Alpha.Mask[4] and Alpha.Mask[19].The board will display "`e _ _ _ t`" and show the alphabet with these two letters missing.

Coding techniques. Two code segments are worth mentioning here. First, note the Alphabet::find() function on lines 277–282. Line 280 is a complete sequential search written in one line, as a `for` loop with no body. It positions `k`, the index for both Alpha and Errors, on the letter that was guessed. The conditional operator in line 281 returns this position or -1, an error code. This is very compact code, but is well within the bounds of readability.

HangWord::print() also uses a one-line loop and a conditional operator to check the mask and print either a puzzle letter or a dash. In contrast, an if statement is used to test the mask in Alphabet::print() and HangWord::try_letter(). The difference is that the last two functions are using a one-sided conditional; they do something if the condition is true, nothing otherwise. The conditional operator can only be used in symmetric situations where some value is returned whether the condition is true or false.

14.4 RandString Adapts a Reusable Data Structure

The RandString class is an *adapter*. It is derived from a general-purpose container class template, FlexArray, and changes the interface provided by the reusable class to one that is appropriate for this application. One new public function, randword(), is added and one existing function is removed from the interface by an override. This is a typical pattern that is repeated over and over when you use class libraries. The library seldom provides exactly the classes you want, but if you can find something close to your needs, you can add to, modify, or restrict its interface to meet you needs. In this application, the interface must be changed to prevent ordinary sequential access and to provide random word selection.

The class RandString is a vocabulary list from which strings may be randomly selected and removed. The letters that form the words are stored in a StringStore. The StringStore class that was presented in Chapter 8 is reused here with a few corrections. StringStore will "take care of itself" and construct as many Pools as needed to hold the characters in the vocabulary words. To store the string pointers, we use a flexible array (vector) because we need both random access and flexibility. Random access is not available with linked lists, so we must use some sort of an array. We don't use a simple array because the number of words in the vocabulary file is not known at compile time.

14.4.1 The RandString Declaration

We create the RandString class by deriving it from an instantiation of the FlexArray template:

```
class RandString : public FlexArray<char*> {...}
```

We aggregate the StringStore in the resulting derived class, and complete the package by adding functions to do random selection and removal of the puzzle words. By this means, we achieve a flexible, sophisticated data structure with very little new code.

The RandString constructor. A few small things deserve notice here:

- Because this is a derived class, we need a ctor to construct the base class (line 338).

- By default, the RandString constructor creates an initial FlexArray that can hold 100 words. This constructor is called from the Game constructor without an integer argument, so the default size is actually used.
- The default FlexArray size is specified in the .hpp file (line 326) but not in the function definition (line 338). This is correct usage.
- There is one restriction on vocabulary words: they must be shorter than 80 characters, since that is the length of the input buffer in the RandStrings constructor (lines 340, 345).
- On line 347, StringStore::put() is called to store the letters; it returns a pointer to the first letter in the new word. This pointer is sent to FlexArray::put(), to be stored in the word array. Very concise, very efficient. Perhaps difficult to understand.
- This is a typical eof-controlled input loop. The break on line 346 will happen if either a read error or and end-of-file occurs. In a game program like this, it is good enough to end input if a read error occurs. They are rare and we can play the game even of some of the words in the file remain unread.

```
310  // ============================================================================
311  // Declaration for a string array with random selection
312  // A. Fischer, May 13, 2001                                 file: rstrings.hpp
313
314  #include "flexT.hpp"
315  #include "sstore.hpp"
316  // ============================================================================
317  class RandString : public FlexArray<char*> {
318    protected:
319      StringStore Store;                  // Storage behind string array.
320
321    private:
322      inline char*  remove( int r );
323      void print( ostream& outs ) const;  // For debugging, make this public.
324
325    public:
326      RandString( istream& vocin, int sz = 100 );
327      ~RandString(){}
328      const char* randword();
329      const char* operator[] ( int index );
330  };
```

The functions randword() and remove(). Random selection and removal of a string is implemented by the randword() function, as follows:

- The RandString constructor, primes the standard C random number generator by calling srand(time(NULL)). This uses the current time of day as an initial value for the randomizing computation, ensuring that different games will start with different puzzle words.
- Game::play() calls RandString::randword() before constructing a new playing board (line 105).
- To select a random word from the N unused words in the vocabulary, randword() generates a random number R in the range $0 \ldots N-1$, then calls remove() to remove the selected word from the vocabulary.
- Inside remove(), the string pointer in the Rth position is copied into a local temporary and later returned. Then it is replaced in the FlexArray by a copy of the last string pointer in the array. Finally, N, the number of words in the vocabulary is decreased by 1. No actual words change position; only pointers are copied. This is a standard algorithm for "shuffling" a deck of cards or randomizing the order of the objects in any array. The strategy is simple, fast, and theoretically sound. It does leave a meaningless copy of a string pointer at the end of the vocabulary array each time a word is used and the array is shortened. Thus, the number of items remaining in the vocabulary, not its original size, must be used to select the next random word.

```
331  // ============================================================================
332  // Implementation for a string array with random selection
333  // A. Fischer, June 4, Nov 14, 2000                         file: rstrings.cpp
334
335  #include "tools.hpp"
```

```
336  #include "rstrings.hpp"
337  // ============================================================================
338  RandString::RandString( istream& vocin, int sz ) : FlexArray<char*>(sz) {
339      //cerr << "\nConstructing RandString  ";
340      char line[80];                      // input buffer
341      srand( time( NULL ) );              // start up random number generator.
342
343      for(;;) {
344          vocin >> ws;
345          vocin.getline( line, 80 );
346          if (!vocin.good()) break;
347          put( Store.put(line, vocin.gcount())); // Add to SStore & FlexArray.
348      }
349      //cerr << "\nRead " <<Many <<" Data from vocabulary file " << endl;
350      if ( !vocin.eof() ) fatal( "Read error on vocabulary file" );
351  }
352  // ----------------------------------------------------------------------------
353  void
354  RandString::print( ostream& outs ) const {
355      outs << "The vocabulary: \n";
356      for (int k=0; k<N; k++) outs << Data[k] << endl;
357  }
358
359  // ----------------------------------------------------------------------------
360  const char*
361  RandString::randword() {
362      if (N < 1) fatal( "Sorry, out of Data!");
363      int r = rand();
364      return remove( r % N );
365  }
366
367  //-----------------------------------------------------------------------------
368  inline char*
369  RandString::remove( int r ) {
370      char*  ret = Data[r];       // Grab the word that was selected.
371      Data[r] = Data[--N];        // Replace by last word in array.
372      return ret;                 // Return word and decrease word count.
373  }
374
375  // ----------------------------------------------------------------------------
376  const char*             // Override; block access to function in base class.
377  RandString::operator[] ( int index )
378  {
379      cerr <<"No random access to vocabulary list";
380      return "";
381  }
```

Overriding FlexArray::operator[]. When using class derivation, all properties of the base class are inherited by the derived class. Sometimes this is undesirable or destructive in the new context. For example, the FlexArray class provides a general subscript operator. However, the RandString class has a very special access and removal rule, and random-access subscripting would defeat its purpose of restrincting access. This problem is handled by *overriding* the inherited definition by a new definition with exactly the same parameters as the inherited method. We cannot eliminate the subscript operator altogether, but we can write it as a trap. In this class, we use a nonfatal trap: it prints an error message but does not abort the run. Clearly, this will never happen in a fully debugged program. However, during construction and debugging, traps like this can be very useful.