Chapter 9: Array Data Structures

A fundamental principle of good construction and good programming:

Function dictates form. First, get the right blueprints.

This chapter covers some major array-based data structures that have been or will be used in the sample programs and/or assignments in this course. These are:

1. Arrays of objects and arrays of pointers.
2. The resizeable array.
3. The stringstore, an efficient way to store dynamically allocated strings.
4. The hashtable.

At the end of the chapter, these data structures are combined in a hashing program that uses an array of pointers to resizeable arrays of string pointers.

General Guidelines. Common to all data structures in C++ are some fundamental rules and guidelines about allocation and deallocation. Here are some guidelines, briefly stated: **The method of deallocation should imitate the method of creation.**

- If you allocate memory with new (and no []), you should free memory with delete.
- If you allocate memory with new in a loop, you should free memory with delete in a loop.
- If you allocate memory with new and [], you should free memory with delete [].

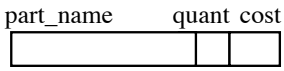
Basic Rules. The guidelines are derived from more fundamental rules about how C++ memory management works:

- If you allocate memory with a declaration, is is allocated on the run-time stack and is freed automatically when control leaves the block that contains the declaration.
- Dynamic memory is allocated using `new` and freed using `delete`. It is allocated in a memory segment called "the heap". If it is not freed, it continues to exist in the heap until program termination.
- If the base type of an array is a class type, you must use delete[] to free it. This automatically calls the base-type destructor function for each element of the array, to free the dynamic extensions attached to it.
- As a general habit, use delete [] to free arrays, even when the [] are not needed. Some programming tools and IDE's become confused when you omit the []. Note, however, that if the base type of an array is a pointer type or a non-class type, it does not matter whether you use delete or delete[] because there are no destructors to run. Dynamic memory attached to the elements of an array of pointers must be freed by explicit calls on delete.
- In one common data structure, the flexible array, an array "grows" by reallocation, as needed. After growth, the array contents are copied from the old array to the new, longer array. In this case, you must free the old array but you must not free anything more than that, so delete must be used without the brackets.

Which should I use? A fundamental dat modeling decision is whether to create the program's objects using declarations (stack allocation) or using pointers and dynamic (heap) allocation. Even in a simple data structure, such as an array of structures, four basic combinations of dynamic and stack allocation are possible, and all are commonly used. We look at these next.

9.1 Allocation and Deallocation of Arrays.

In this section, we will be using arrays of a representative class type named Item, which has three data members, as shown below.




```
class Item {
    char part_name[20];
    int quant;
    float cost;
  public:
    Item();
    ~Item();
};
```

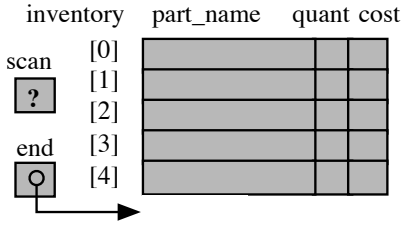
Four array data structures are presented, all of which implement some form of two-dimensional structure. However, they have different storage requirements and dynamic possibilities. It is important for you to understand how they differ theoretically and practically, how to create and use each one, and when each one might be preferred.

A diagram is given of the storage allocated for each data structure. Core portions of these objects are allocated on the run-time stack and are colored gray. Extensions are allocated in the heap and are colored white. To the right of each diagram is a code fragment that shows how to use new and delete to create and deallocate the structure.

The first data structure given is a simple array of Items whose size is fixed at compile time. Each succeeding example adds one level of dynamic allocation to the data structure, until the last is fully dynamic. For the sake of comparison, these four data structures are implemented as uniformly as possible. All are five slots long and have offboard end pointers to assist in sequential array processing. (Note: inventory+5 is the same as &inventory[5].)

1. **A declared array.** This data structure is used if the length of the array is known at the time control enters the block containing the declarations. All storage is allocated by the array declaration, so no "new" is necessary. For an array of class objects, a default constructor must be provided and will be used to initialize the array.

   No "delete" is necessary and using "delete" or "delete[]" is a fatal error because this storage will be automatically deallocated at the end of the function that contains the declarations and storage must not be deallocated twice.

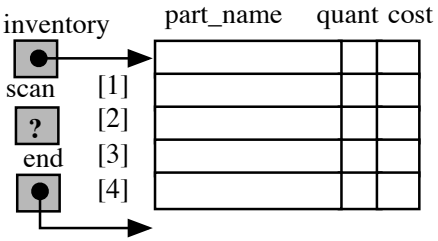



```
Item  inventory[5];          // Allocate array.
Item* end = &inventory[5];   // Connect end pointer.
Item* scan;                  // Processing pointer.
```

2. **A single dynamic array of class objects.** This data structure is used if the length of the array is not known at compile time but is known at run time, before any data is stored in the array. It could be combined with automatic reallocation to store data sets that might grow larger than the initially allocated size. However, this requires that all data in the array be copied into the reallocated memory area, a potentially large cost.

   The core portion of this object is created by declaring two Item pointers. The extension is created by a call on "new", often in the constructor for some class that contains the array. A default Item constructor must be present and will be used to initialize the array. Dynamic arrays of non-class base types cannot be initialized.

   Since "new" was used with [] to create this array, "delete[]" must be used to deallocate it.

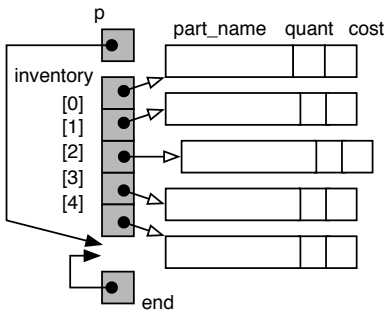



```
Item* inventory;          // Head pointer (core).
Item* end;                // Sentinel pointer (core).
Item* scan;               // Processing pointer (core).

inventory = new Item[5]; // Allocate array (extension).
end = &inventory[5];      // Connect end pointer.

delete[] inventory;       // Delete the array.
```

3. **A declared array of pointers to individual dynamic objects.** This data structure can be used if the Item constructor needs parameters or if the maximum amount of storage needed is predictable, but you wish to avoid allocating the storage until you know it is needed. An array of Item pointers is pre-allocated and initialized to NULL. A new Item is allocated and attached to the data structure each time new storage is needed. Any constructor (not just a default constructor) can be used to initialize these items because they are individually created. Since "new" was used without [] to create the Items, "delete" must be used without [] to deallocate them. Since a loop was used for allocation, a loop must be used for deallocation.

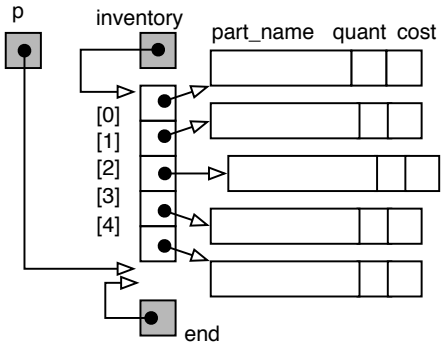



```
Item* inventory[5];                    // Create core.
Item** end = &inventory[5];            // Tail sentinel.
Item** p;                              // Scanner.

for (p=inventory; p<end; ++p) {        // Allocation.
    // Read and validate data.
    *p = new Item(field_1...field_n); // Install data.
}
for (p=inventory; p<end; ++p) {
// Deletion loop.
    delete *p;
}
```

4. **A dynamic array of pointers to individual dynamic objects.** This data structure is used when the amount of storage needed is unpredictable. An Item** is declared and initialized at run time to a dynamic array of Item pointers, each of which should be initialized to NULL. By using a flex-array data structure, the array of pointers can be easily lengthened, if needed. The cost of the reallocation and copy operations is minimal because only the pointers (not the Items) need to be copied.

   Another Item is allocated and attached to the data structure each time new storage is needed. Any constructor (not just a default constructor) can be used to initialize these items because they are individually created. Since "new" was used without [] to create the Items, "delete" must be used without [] to deallocate them. Since a loop was used for allocation, a loop must be used for deallocation. After the deallocation loop, the main array (the backbone) must also be freed by using delete with [].




```
Item** inventory = new Item*[5];       // Allocate *s.
Item** end = &inventory[5];            // Tail pointer.
Item** p;                              // Scanner.
for (p=inventory; p<end; ++p) {        // Allocation.
    // Read and validate data.
    *p = new Item(field_1...field_n);  // Install data.
}

for (p=inventory; p<end; ++p) delete *p;
delete[] inventory;
```

9.2 The Flexible Array Data Structure

A flexable array is a container class, that is, a class whose purpose is to contain a set of objects of some other type. The C++ standard template library (formerly called STL) contains a template class, named `vector`, from which a flexible array of any type may be created. Java supports a type named `vector` and another named `arraylist`, that are basically similar. All of these classes are based on a dynamically allocated array that is automatically reallocated, when necessary to contain more data. Class functions are used to store data in the container and retrieve it.

A flexible array is a good alternative to a linked list for applications in which an array would be appropriate and desirable but the amount of storage needed is unpredictable. It works especially well when sequential (not random) access is used to add data to the array, as in the simple program below. In this section, we present a simple version named FlexArray, which does not rely on the use of templates. We then use a FlexArray to create an array of characters that can hold an input string of any length.

9.2.1 Implementation in C++

The flexible array data structure consists of a pointer to a long dynamically-allocated array for data, an integer (the current allocation length), and the subscript of the first unused array slot. When there is data to be stored in the container, but it is full, the flexible array automatically doubles in length. The cost is a modest amount of time spent reallocating and copying storage. This can easily be less than the time required to manipulate links and traverse lists. Wasted space is minimal for a flexible array, often less than for a linked structure, but the major advantage of the flexible array is that a programmer can use array-based algorithms (quicksort) and random access to process the data in the array.

The class FlexChars is an implementation of a flexible array with base type char. It is a good example of how a class can (and should) take care of itself. It has one private function, `grow()` (lines 14 and 68–76), which is called whenever there is a object to store in the FlexChars, but the array storage space is full. The existence and operation of this function is completely hidden from the client program; the data structure simply "grows" whenever necessary.

Before enlarging the array, we test the value of Max. If it is a legal array length, it is doubled; otherwise it is set to the default length. The array is then reallocated at the new, longer, length, the data is copied from the old array into the new one, and the old one is freed.

The FlexChars class declaration.

- Lines 17–21 define the constructor. Note that the parameter `ss` has a default value of `FLEX_START`. This permits a client program to call the FlexChars constructor with or without an argument.
- Line 22 is the destructor. Note that it tests for a NULL pointer before attempting to free the array, and it uses `delete[]` because the corresponding constructor used `new` with `[]`.
- Two "get" functions, `length()` and `message()` (lines 28 and 29) provide read-only access to the number of chars actually stored in the flexible array and to the character string itself..
- A print function is (lines 31...34) allows the FlexChars client program to print the contents of the array without knowing how it is implemented.
- The `put` function (lines 25 and 61–66) allows us to store information in the next free slot of the FlexChars. It checks for available space, calls `grow()` if necessary, stores the data, and increments `N`, the address of the first available slot. This function provides the only way to store new data in the FlexChars.
- The `grow` function (lines 14 and 68–76) is the heart of this class. It is called from `put` when the array is full and more input must be stored. The diagrams on the next page illustrate its operation, step-by-step.
- The extension of the `subscript` operator (lines 26 and 55–59) is not used in this application but will be used later in the chapter. It will be explained in Chapter 10.

```
1  //-------------------------------------------------------------------------
2  // Class declaration for a flexible array of base type char.
3  // A. Fischer, February 7, 2003                        file: flexChars.hpp
```

```
4   #ifndef FLEXC
5   #define FLEXC
6   #include "tools.hpp"
7   #define CHARS_START 4       // Default length for initial array.
8
9   class FlexChars {
10    private: // ------------------------------------------------------------
11      int Max;        // Current allocation size.
12      int N;          // Number of array slots that contain data.
13      char* Data;     // Pointer to dynamic array of char.
14      void grow();    // Double the allocation length.
15
16    public: // ------------------------------------------------------------
17      FlexChars( int ss = CHARS_START ) {
18          Max = ss;               // Current allocation length
19          N = 0;                  // Number of data items stored in array.
20          Data = new char[Max];    // An array to contain objects of type char.
21      }
22      ~FlexChars() { if (Data != NULL) delete[] Data; }
23
24      void inputLine( istream& ins );
25      int  put( char data );       // Store data and return the subscript.
26      char& operator[]( int k );
27
28      int  length() { return N; }  // Provide read-only access to array length.
29      const char* message() { return Data; }  // Read-only access to the chars.
30
31      ostream& print(ostream& out) { // Print the filled part, ignore the rest.
32          Data[N] = '\0';
33          return out <<Data;
34      }
35  };
36  inline ostream& operator<< (ostream& out, FlexChars& F){ return F.print( out ); }
37  #endif
```

The FlexChars class implementation.

```
38  //-----------------------------------------------------------------------------------
39  // Implementation of the FlexChars class.
40  // A. Fischer, February 7, 2003                                    file: flexChars.cpp
41
42  #include "flexChars.hpp"
43  void  FlexChars ::   // ------------------- Read and return one line from input stream.
44  inputLine( istream& ins ){                  // Discard prior contents of the buffer.
45          char ch;
46          for (N=0; ; ){                      // Start with slot 0 of the buffer.
47              ins.get( ch );
48              if (!ins.good() || ch == '\n') break;  // Go until eof or newline.
49              put( ch );                      // The put function takes care if itself.
50          }
51          put( '\0' );                        // Terminate the string in the buffer.
52          //cerr <<"Input length = " << N <<endl;
53  }
54
55  char& FlexChars :: //------------------------------- Access the kth char in the array.
56  operator[]( int k ) {
57      if ( k >= N ) fatal("Flex_array bounds error.");
58      return Data[k];                          // Return reference to desired array slot.
59  }
60
61  int FlexChars :: // ------------------------------------- Copy a char into the array.
62  put( char ch ) {
63      if ( N == Max ) grow();     // Create more space if necessary.
64      Data[N] = ch;
65      return N++;                 // Return subscript at which item was stored.
66  }
```

```
67
68  void FlexChars :: // ---------------------------------- Double the allocation length.
69  grow() {
70      char* temp = Data;                      // hang onto old data array.
71      Max>0 ? Max*=2 : Max = CHARS_START;
72      Data = new char[Max];                   // allocate a bigger one.
73      memcpy(Data, temp, N*sizeof(char));     // copy info into new array.
74      delete temp;                            // recycle (free) old array.
75                  // but do not free the things that were contained in it.
76  }
```

The main program. The main program for this example shows a simple application for a flexible array: reading a line of input whose length is not known ahead of time. The main program creates a FlexChars named `sentence` (line 87) and uses it to read one line of input. The input is then echoed, with its length, as shown below:

```
77  //=========================================================================
78  // Using a FlexChars of characters.
79  // A. Fischer, May 4, 2002                              file: flex_test.cpp
80  //
81  #include "tools.hpp"
82  #include "flexChars.hpp"
83  //-------------------------------------------------------------------------
84  int main( void )
85  {
86      banner();
87      FlexChars sentence; // A FlexChars of 20 chars.
88
89      cout <<"\nFlexible Array Demo Program \nEnter a long sentence:\n";
90      // Read an entire line of text; the buffer will grow when necessary.
91      sentence.inputLine( cin );
92
93      // End of input phase -- ready for output.
94      cout <<"\nSentence length is " <<sentence.length() <<endl;
95      cout << sentence.message() <<endl;  // Print an ordinary string.
96  }
```

The output:

```
Flexible Array Demo Program
Enter a long sentence:
Today is May 4.  It has been a lovely day outside -- I planted some flowers and enjoyed the soft,
warm, spring air.   Now evening is coming and the sun shines only on the treetops.

Sentence length is 181
Today is May 4.  It has been a lovely day outside -- I planted some flowers and enjoyed the soft,
warm, spring air.   Now evening is coming and the sun shines only on the treetops.
```

Tracing the operation of the FlexChars. The FlexChars data structure created by `main()` is illustrated in five stages of growth during the input operation. The object named sentence is shown on the left side of Figure 8.1, just after the constructor is done initializing it (line 21). The initial length of the array is `STARTSIZE`, or 4 characters.

Inside the `inputLine()` function, a loop is used to read characters one at a time and put them into the flexible char array. (A newline or eof condition will end the loop.) The right side of Figure 8.1, shows `sentence` again after four characters have been put into it, making it full.

When the command to put a fifth character into `sentence` is given, the FlexChars must grow. (Figure 8.2, left). This is done in the `grow()` function. In the C++ implementation, the Max length is doubled and a new data array of the new, larger length is allocated. then the data is copied from the old array to the new and the old array is deleted. Finally, the fifth character is stored in the newly-lengthened array.

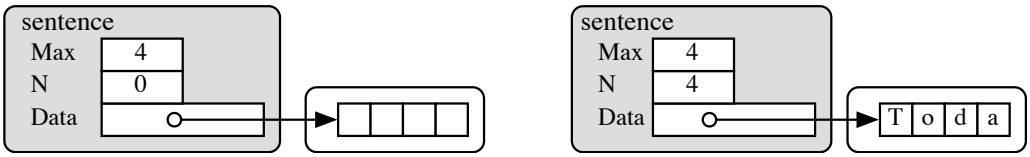


Figure 9.1: An empty FlexChars object (left) and one that is ready to grow (right).

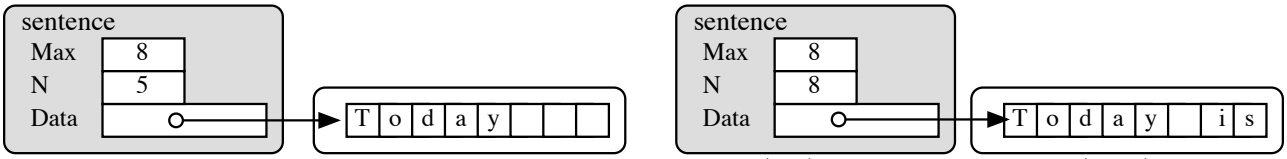


Figure 9.2: A FlexChars object after first growth spurt (left) and ready for second (right).

Input then continues normally, filling the rest of the slots (Figure 8.2, right). When the command is given to store the ninth character, the FlexChars must grow again. The doubling process is repeated, resulting in an array of 16 characters. The ninth character is then stored in the new space and input continues. Figure 8.3 shows the array after the tenth character is input, with ten slots in use and space for six more letters.

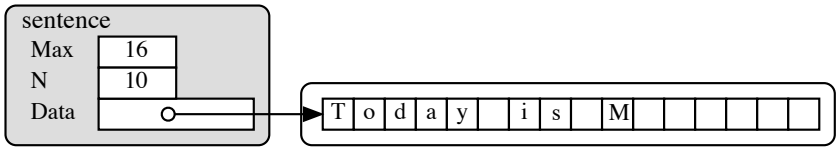


Figure 9.3: The same FlexChars object after doubling the second time.

Costs. This is an acceptably efficient algorithm. After the first doubling, the number of array slots allocated is always less than twice the number in use, and the total number of objects that have been copied, including all copy operations, is slightly less than the current length of the array.

Caution. Because the entire array is moved during the growth process, any pointers that point *into* the array must be re-attached after the reallocation. Because of this, a flexible array is not an appropriate data structure when many pointers point into an array. Pointers pointing *out of* the reallocated array are not a problem.

9.2.2 Implementation in C

In C, the doubling is accomplished using the `realloc()` function:

```
data = (char*)realloc( data, new_size );
```

This function takes any memory area that had been created by `malloc`, `calloc` or `realloc` and changes its size to the new size requested. If the new size is shorter, the excess memory is returned to the system for future re-use. If the new size is longer and the adjacent memory area is free, the existing area is simply lengthened. Otherwise, a new, larger area is allocated somewhere else in memory and the contents of the array are copied into the new area. This is easier and more efficient than the C++ implementation, in which the reallocation and copying must be done step-by-step.

9.3 Ragged Arrays

A ragged array is a convenient data structure for representing any list of phrases such as a menu, labels for a graph, or error comments. It has two advantages over the traditional 2-dimensional array of characters.

- Since the length of the strings in the ragged array can vary, the space they take is kept to a minimum. (Each phrase requires one byte per letter, one for the '\0', and four bytes of overhead.)
- Dynamic modification of the data structure is possible. The strings can be allocated and connected as needed and can be repositioned by swapping pointers.

Figure 8.4 illustrates the two kinds of ragged array: a constant version built from literals, and a dynamic version that can contain user input.

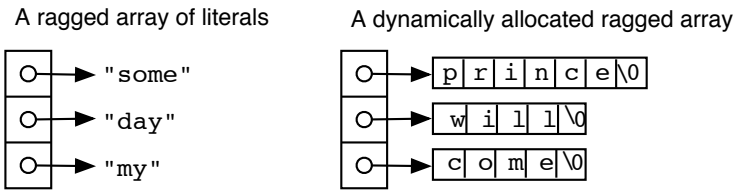


Figure 9.4: Two kinds of ragged arrays.

A literal ragged array is most easily constructed by a declaration with a list of literals as the initializer:

```
char* vocabulary[] = { "Some", "day", "my" };
```

9.3.1 Dynamic Ragged Arrays

A dynamic ragged array is straightforward to build. Each "rag" is constructed separately and attached to a slot in a "backbone" array, which could be either declared or allocated dynamically. Several steps are required to construct a rag, as shown by the `getRag()` function:

```
97  // ------------------------------------------------------------------------
98  // Read one string and store in dynamic memory.             file: getRag.cpp
99  //
100 #include "tools.hpp"
101
102 char* getRag( istream& fin )
103 {
104     static char buf[80];        // Input buffer limited to 80 chars.
105     int len;                    // Length of input phrase.
106     char* item;
107
108     fin.getline( buf, 80 );     // Don't read past end of buffer.
109     len = fin.gcount();         // Measure the input string.
110     item = new char[len];       // With getline, len is always >= 1.
111     strcpy( item, buf );        // Copy input into the new space.
112     return item;                // Return dynamic storage containing input.
113 }
```

Attaching the rags. The following main program uses `getRag()` in a common way: to create a list of options from a file. This approach is better than building the menu into a program because it allows the menu to be changed easily.

For the backbone of the data structure, this main program uses an array with a fixed maximum size (line 124). However, a flexible array could be used and would often be a better choice. The main loop calls `getRag()` repeatedly (line 132) and attaches the resulting rags to the backbone. Filling all the array slots (line 131) or an end-of-file condition (line 133) ends the input loop. Finally `main()` prints the complete list (line 135).

Note that `main()` is the function that tests for eof, not `getRag()`. This is a common pattern in C++ because many input functions read only a single data set and deliver the resulting single object to a client program. The input function takes a stream as a parameter and does not know what kind of stream that might be: an istream or an ifstream or a istringstream.

In contrast, the calling function must get and store many inputs. It knows where the inputs come from and how many can be handled. Therefore, the calling function has the responsibility for ending an input loop.

```
114 #include "tools.hpp"
115 #define INPUT "menu.in"
116
117 char* getRag( istream& );
118 void print_menu( char*[], int );
```

```
119
120  // ---------------------------------------------------------------
121  int main (void)
122  {
123      int k;                          // Line counter.
124      char* menu[20];                 // Backbone of ragged array.
125
126      cout <<"Ragged Array Demo Program\n";
127      ifstream menufile( INPUT );     // Open file named by define.
128      if (!menufile) fatal( "Cannot open file %s." INPUT );
129
130      cout <<"Reading menu items from " <<INPUT <<"\n";
131      for(k=0; k<20; ++k)  {
132          menu[k] = getRag( menufile );
133          if (menufile.eof()) break;  // User decided to quit.
134      }
135      print_menu( menu, k );
136      return 0;
137  }
138
139  // ---------------------------------------------------------------
140  void print_menu( char* menu[], int n )
141  {
142      int k;                          /* subscript variable for loop */
143      cout <<"\n" <<n <<" skating sessions are available:\n";
144      for (k = 0; k < n; ++k)  cout << '\t' <<menu[k] <<endl;
145  }
```

9.4 The StringStore Data Structure

Each time we allocate space dynamically, the C and C++ systems allocate extra bytes to store the length of the allocation. This can use a significant amount of storage if many separate small allocations are done. A StringStore is a more efficient way to store an unlimited number of variable-length strings, such as might be needed for a dictionary or a long menu. This data structure consists of one or more long dynamically allocated arrays called "pools", each of which will hold many strings. When the first pool is full, another is allocated. These pools are linked together to facilitate deallocation at the end of their useful lifetime. Figure 8.5 shows the first pool allocated for a stringstore, pointing at NULL. The oldest pool will always be at the end of the list of pools; the newest pool will be at the head of the list.

A StringStore has three fields: a pointer to the beginning of the most recently allocated pool, the amount of space remaining in that pool, and the subscript of its first unused slot. Figure 8.6 shows a StringStore with one Pool.

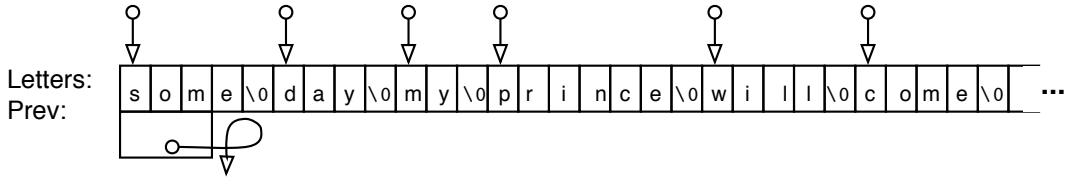


Figure 9.5: A Pool that contains 6 words.

9.4.1 The StringStore and Pool Classes.

```
1   //========================================================================
2   // Class declarations for StringStore and Pool.
3   // A. Fischer, June 4, 2000                              file: sstore.hpp
4   #ifndef SSTORE
5   #define SSTORE
6
7   #include "tools.hpp"
```

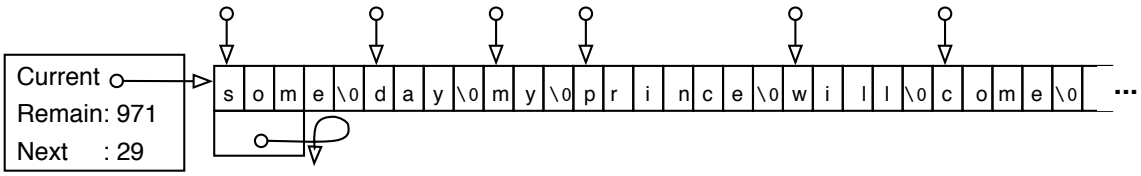


Figure 9.6: A StringStore with one Pool.

```
 8  #define MAXLETS  1000   // Pool size.
 9  //========================================================================
10  class Pool {
11    friend class StringStore;
12    private:
13      char  Letters[MAXLETS]; // Memory behind stringarray.
14      Pool* Prev;             // Previous StringStore, for deallocation.
15
16      Pool( Pool* pv = NULL ) { Prev = pv; }
17      ~Pool(){}
18  };
19
20  //========================================================================
21  class StringStore {
22    private:
23      Pool* Current;  // Pool that is currently being filled.
24      int Remain;     // Space remaining in current Pool.
25      int Next;       // Subscript of next available slot.
26
27    public:           //-------------------------------------------------
28      StringStore( int sz = MAXLETS ): Current(new Pool), Remain(sz), Next(0) {}
29      ~StringStore();
30      char* put( const char* s, int len );
31  };
32  #endif
```

The advantage of this structure over individually-allocated strings is that both space and time are used efficiently. There are fewer calls for dynamically allocation, so there are fewer areas to delete later and less space wasted in overhead. The disadvantage is that, if strings are deleted out of the middle of the array, the space cannot be easily reclaimed or reused. Thus, the data structure is appropriate for relatively stable lists of alphabetic information.

The StringStore interface. A StringStore serves only one purpose: to store a large number of strings efficiently, on demand. The strings are not accessed through functions of the StringStore class. Instead, the class that calls StringStore::put() receives a pointer to the stored string and provides access to the string through that pointer. Thus, the only functions that are needed in this class are a constructor, a destructor, and the put function. During debugging, it might help to have a print function, but that would never be called by a client class.

Each StringStore class has one or more Pools associated with it. A one-many relationship of this sort is usually implemented by having the primary class point at a list of objects of the secondary class. Neither class makes sense without the other; they form a tightly coupled pair. There are two ways a tightly-coupled pair of classes can be implemented: the secondary class can be defined inside the primary class, or the secondary class can be declared to be a friend of the primary class. I dislike placing the definition of the helper class inside the interface class because it is wordy and syntactically awkward.

I prefer using friendship. It is easier to write and clearer to read. Friendship gives the primary (or interface) class unrestricted access to all of the members of the secondary (or helper) class. The helper class only needs constructor and destructor functions; all other functionality can be handled by the primary class. In my StringStore definition, the Pool class gives friendship to the StringStore (line 11).

```
33  //========================================================================
```

```
34  // StringStore data structure -- implementation.
35  // A. Fischer, May 29, 2000                                  file: sstore.cpp
36
37  #include "sstore.hpp"
38  //---------------------------------------------------------------------------
39  StringStore::~StringStore() {
40      Pool* p;
41      while (Current != NULL) {
42        p = Current;
43        Current = p->Prev;
44        delete p;
45      }
46  }
47
48  //---------------------------------------------------------------------------
49  char*
50  StringStore::put( const char* s, int len ) {
51      char* where;
52      len++;                              // number of characters including '\0'.
53      if ( len > Remain ) {               // no room to store s in current Pool.
54          Current = new Pool( Current );  // Attach new pool at head of chain.
55          Next = 0;                       // New pool is empty.
56          Remain = MAXLETS;               // All slots are unused.
57      }
58      where = &Current->Letters[Next];    // Number of characters including '\0'.
59      Next += len;                        // Move pointer to next empty slot.
60      Remain -= len;                      // Decrease number of remaining bytes.
61      strcpy( where, s );                 // Copy the new string into the array.
62      //cerr << where <<" ";
63      return where;                       // First letter of newly stored string.
64  }
```

The StringStore implementation. Internally, the StringStore and pool classes follow the usual form for a linked list. The interface class, StringStore, points at a linked list of pools, and the pool class gives friendship to StringStore. Each pool contains a long dynamically allocated char array and points at the previously allocated pool (or NULL). Each char array is long enough for 1000 characters, and we expect to be storing words or phrases that vary from a few to about 30 characters. Thus, we expect to waste less than 15 chars at the end of each pool array, a modest amount.

As strings are read in by a client program, they are stored by calling StringStore::put(). This function copies the string into the next available slot of the current pool, updates `next` and `remain`, and returns a pointer to the beginning of stored string. The client must store this pointer in some other data structure such as a linked list, array, flex_array, or hash_table. If the space remaining in the current pool is not large enough to hold the current input string, another pool is allocated and attached to the head of the list of pools. The length of the pools should be great enough so that the space wasted at the end of each pool is only a small portion of the space allocated. One of the pointers at the top of this StringStore diagram will be returned each time StringStore::put() is called. The caller must store the pointer in some data structure such as an array or a linked list.

9.5 The StringArray

A StringStore can be used with any data structure that stores string pointers. Of these, the simplest is the StringArray, shown below, which is simply an array of string pointers. This data structures has two strong advantages over a linked list of words: the elements of an array can be searched and sorted much more efficiently than the same data stored in a linked list. In the diagram below, we show a StringArray named `SA`, which has a StringStore named `Letters` as its first member. The second member is an array of string pointers named `Words`. Each of the six items stored in `Words` is a pointer to one of the strings in `Letters`.
The program below illustrates the use of a StringArray to organize a set of pointers to words in a StringStore. It also demonstrates how to use the standard quicksort function (from stdlib.h) to sort that array.

Notes on the StringArray class declaration.

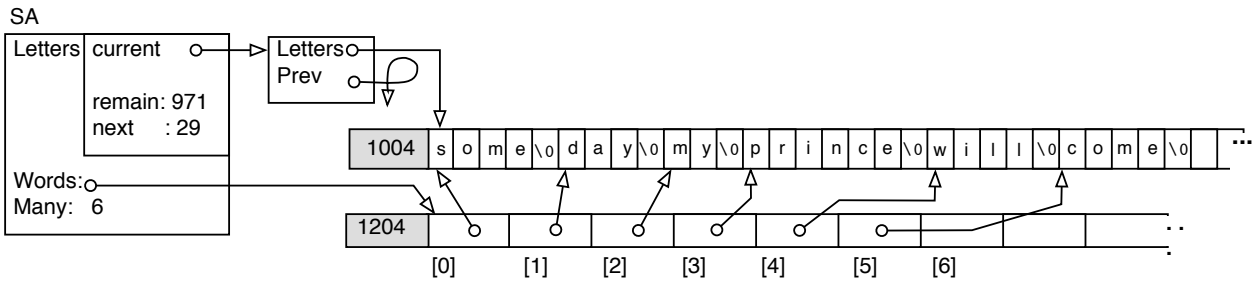


Figure 9.7: A String Array named SA.

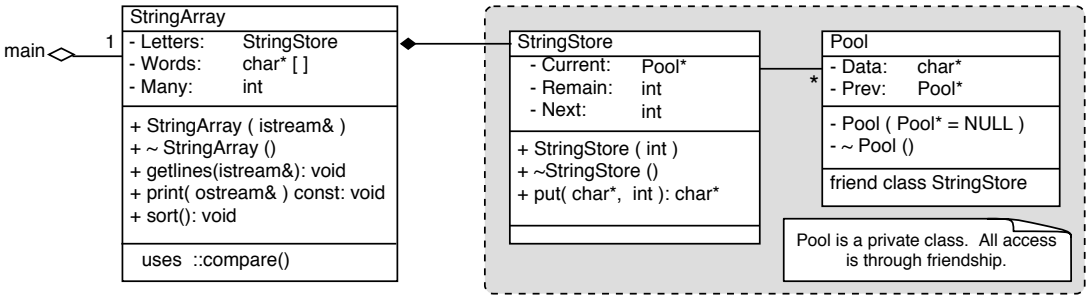


Figure 9.8: UML diagram for the String Array program.

- Three header files are included: `sstore.hpp` is needed because one member of this class is a StringStore. `FlexChars.hpp` is needed because we will use a flexible array as an input buffer. `compare.hpp` is needed because one of the StringArray class functions is `sort()` that cannot be part of the class because it must interact with which `qsort()`, a standard C library function.

- This implementation restricts the word list to 300 words. This is an arbitrary maximum, large enough to test the code but not large enough to be useful for most word lists. The `Vector` class or a flexible array of strings could be used to remove this limitation.

- The sort function (Line 87) is defined as a class member because it must refer to the data members of the class. It sorts the strings by calling `qsort()`. The arguments are (1) the array to be sorted, (2) the number of data items stored in that array, (3) the size of the base type of the array, and (4) a function that `qsort()` will call many times to compare pairs of array elements. The `compare()` function is a global function in the file `compare.hpp`.

- Line 85 is a do-nothing destructor. It is good style to define a destructor for every class, even if it does nothing.

```
65  //=============================================================================
66  // Class declaration for StringArray: an array of words using a StringStore.
67  // A. Fischer, October 3, 2000                                file: sarray.hpp
68  //
69  #ifndef SARRAY
70  #define SARRAY
71
72  #include "flexChars.hpp"
73  #include "sstore.hpp"
74  #include "compare.hpp"
75  #define MAXWORDS 300            // Limit on number of words in vocabulary.
76
77  //-----------------------------------------------------------------------------
78  class StringArray {
79    private:                      //-----------------------------------------
80      StringStore Letters;        // Storage behind string array.
81      char* Words[MAXWORDS];      // String array.
```

```
82      int Many;                   // Number of words and first available slot.
83    public:                       //-----------------------------------------
84      StringArray( istream& in );
85      ~StringArray(){}
86      void print( ostream& out ) const;
87      void sort(){ qsort( Words, Many, sizeof(char*), compare ); }
88  };
89  #endif

91  //============================================================================
92  // Implementation of StringArray: an array of words using a StringStore.
93  // A. Fischer, February 8, 2003                              file: sarray.cpp
94
95  #include "sarray.hpp"
96  // ---------------------------------------------------------------------------
97  StringArray::StringArray( istream& sin )
98  {
99      Many = 0;                   // Array is empty; the first available slot is [0].
100     const char* word;
101     for (;;) {
102         FlexChars buf;          // Allocate a new flexible char array for the input.
103         buf.inputLine( sin );   // A flexible buffer has no fixed maximum length.
104         if ( sin.eof() ) break;
105         word = buf.message();
106         if ( word[0] != '\0') {
107             Words[Many++] = Letters.put( word, buf.length() );
108             //cout << "\n> " << word <<"\n";
109         }
110     }
111 }
112
113 // ---------------------------------------------------------------------------
114 void StringArray::print( ostream& outs ) const
115 {
116     for (int k=0; k<Many; k++) cout << Words[k] << endl;
117 }
```

Notes on the StringArray class implementation.

- Line 103 reads the input into a temporary buffer; it is later put into a more efficient permanent data structure. Use of a flexible array for an input buffer enables us to safely see, validate, and measure any input string before we need to allocate permanent storage for it.
- Line 104 tests for the end of the file and breaks out of the loop if it is found. This is both correct and commonly done, even though some other function did the actual input.
- Line 107 copies the input word from the temporary buffer into the StringStore and saves the resulting pointer in the StringArray.
- The use of postincrement on the left side of line 107 is the normal and usual way to store a value in an array and update the array index to be ready for the next input.
- Line 108 is commented out; it was useful when I debugged this class.
- The `const` on lines 86 and 114 indicates that the `print()` function does not change the values of any class members. This is useful as documentation and as insurance that you have not accidentally assigned a value to a class member.

The compare() function.

```
118  #ifndef COMPARE
119  #define COMPARE
120  // ------------------------------------------------------------------------
121  // Called by qsort() to compare two strings (non-class objects).
122  // To compare two class objects, the last line would call a class function.
123  //
124  inline int
125  compare( const void* s1, const void* s2 )
126  {
127      char* ss1 = *(char**)s1;
128      char* ss2 = *(char**)s2;
129      return strcmp( ss1, ss2 );      // This compares two non-class objects
130  }
131  #endif
```

- This function compares two strings in a way that is compatible with the standard qsort() function. The complexity of the call and the compare function are due to the general nature of `qsort()`: it can sort an array of any base type containing any number of data elements. Several casts and the use of pointer-pointers are necessary to achieve this level of generality.
- The parameters are type void*, which is compatible with any pointer type. The arguments will be pointers to strings, that is, char**. First, we need to cast the void* to type char**, to match the argument. Then, to use strcmp, we need to extract the char* from the char**.
- You can use `qsort()` and write `compare()` functions by following this example, even if you don't fully understand it. To sort a different data type, just change the char* to float or int or whatever you need, and change the strcmp to the appropriate comparison operator or function.

The main program.

```
132  //========================================================================
133  // StringArray, StringStore, and qsort Demo Program
134  // A. Fischer, February 8, 2003                    file: ArrayQsort/main.cpp
135  //========================================================================
136  #include "tools.hpp"
137  #include "sarray.hpp"
138
139  int main ( int argc, char* argv[] )
140  {
141      banner();
142      if (argc < 2) fatal( "Usage: ArrayQsort input_file");
143      ifstream vocabIn( argv[1] );
144      if (!vocabIn) fatal( "\nCould not open file vocab.in" );
145
146      StringArray vocab( vocabIn );
147      cout << "\nFile successfully read; ready for sorting.\n\n";
148      vocab.sort();
149      vocab.print( cout );
150      return 0;
151  }
```

Notes on the main program. This program reads a list of words, prints it, sorts it, and prints it again. The relationships among the main program and all the functions called are illustrated by the call chart below. The main program follows the typical form. It includes only one application-specific header file (for the main data structure). Two objects (a stream and a string array) are declared. All work is delegated to class functions.

The output. The input file (left) and the output (right) have been arragned in two columns below, to fit the space. The banner and bye messages have been omitted from the output.

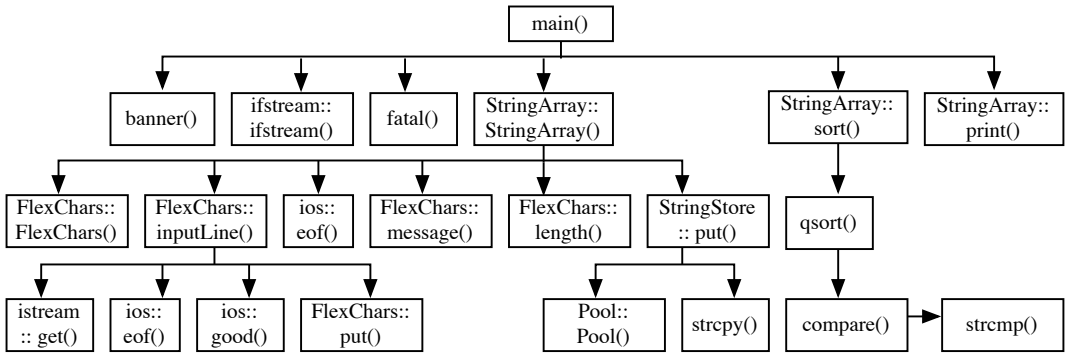


Figure 9.9: A call chart for the sorting example.

```
This is a file of strings.
It should end up sorted.
To demonstrate this, though,
I need several words.
This
is
the
end.
```

```
File successfully read; ready for sorting.

I need several words.
It should end up sorted.
This
This is a file of strings.
To demonstrate this, though,
end.
is
the
```

At the end of this chapter, we implement a Typing Tutor application using a hash table and a FlexString class, which is a built by combining the techniques of the flexible array and the String Array.

9.6 Hashing

A hashtable is a container class that can store and retrieve data items quickly and efficiently but does not keep them in a predictable or sorted order. The position of a data item in the hash table is derived from a key data field, or a key constructed from parts of several data fields. When implementing a hash table, three issues must be decided:

1. The backbone of a hash table is an array. How many slots will it have?
2. Each slot of the hash array is called a bucket. What data structure will you use to implement a bucket?
3. What algorithm should be used for your hash function? To assign an item to a bucket, a table subscript is calculated from the key field. The algorithm that does the calculation is called the *hash function*. It must be chosen to work well the expected data set. *Hashing* is the act of applying the algorithm to a data item's key field to get a bucket number.

In most hashing applications, an Item is a structure with several fields and the key field is just one of many. In such applications, distinct Items may have the same key. In this discussion, we imagine that we are hashing words for a dictionary. Since the Item will be a simple null-terminated string, the entire item will be the key field.

9.6.1 The Hash Table Array

Table structure. There are two basic ways to implement a hash table:

- Each bucket in the main hash array can store exactly one Item. Using this plan, the array must be longer than the total number of Items to be stored in it, and will not function well unless it is nearly twice that long. When a key hashes to a bucket that is already occupied, it is called a *collision* and some secondary strategy must be used to find an available bucket. Often, a second (and possibly a third) function is used; this is called *rehashing*.
- Each bucket in the main hash array can store an indefinite number of Items. Rehashing is not necessary; dynamic allocation takes its place. Ideally, the list of items in a bucket should be short so that it can be

searched quickly. This fact is used in calculating NBUK: if the goal is an average of B items per bucket, and we expect to have N items total, then NBUK should be approcimately N/B.

This kind of hash table is much easier to implement and will be used here. The bucket can be implemented by any dynamic data structure; most commonly used are linked lists, hash tables, and flex-arrays. In this discussion, we use a flex array to implement a bucket, as in Figure 8.6.

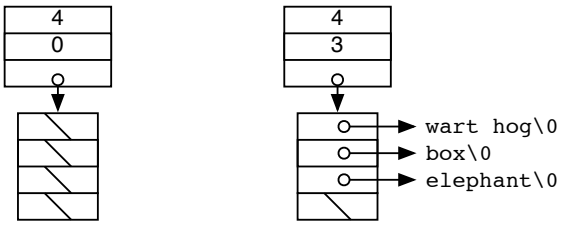


Figure 9.10: An empty Bucket (left) and one containing three Items (right).

Our chosen hash table implementaion is an array of NBUK buckets where each bucket is a flexarray, as shown in Figure 8.7. The buckets will be initialized in the Bucket constructor to be empty flex-arrays with 4 slots of type Item* where items can be attached later.

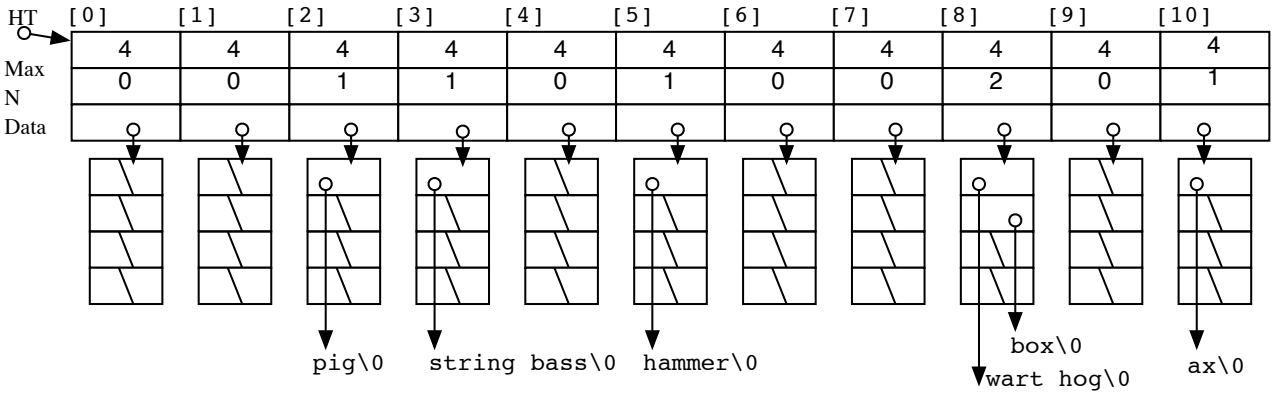


Figure 9.11: A HashTable with 11 Buckets and 6 Items.

Table length. For reasons related to number theory, and beyond the scope of this discussion, hashing works best when the number of the buckets is not a multiple of 2. Normally, the programmer would choose a large prime number for the table length. Here, we choose a small one so that the diagrams will fit on a page. When you implement your own hash table, start with a small number of buckets and do the initial debugging with a modest number of data items. When the program is ready for a full-scale trial, change the NBUK to a large prime.

Inserting a new Item. Suppose that D is a data item and that the program wants to insert it into a hash table. Before insertion, the program should search the table to determine whether it is duplicate data. If not, space whould be allocated for for the new data dynamically. The key field is then hashed to find the correct bucket number, and the FlexArray::put() function is used to insert the resulting pointer at the end of the appropriate list. Because a flex-array is being used, each list will automatically extend itself if it becomes overfull.

Retrieving an Item. To find an item in the table, we must hash its key field and search the correct bucket and search it for the desired key. Since every hash bucket should be fairly short, a simple sequential search is appropriate. If more than one item may have the same key field, more comparisons are needed to confirm whether or not the desired item (or another with the same key) has been located.

9.6.2 Hash Functions.

When choosing a hash function, several things are important.

1. It should use all of the information in the hash key, not just the beginning or end.
2. Its result must be a subscript between 0 and NBUK-1. This is easily accomplished by using the key to generate a larger number and using the *mod* operator (`%`) to reduce it to the desired range.
3. It should be efficient. Hash functions that make a lot of unnecessary calculations should be avoided. Bit operations (bitwise exor and shifts) are fast; addition and subtraction are slower, multiplication is slow, and division and mod are very slow. The last operation in the hash function must be mod, but, with that exception, the slower operators should be avoided. Remember—the object is to calculate a meaningless random number based on the key.
4. It should spread the data as evenly as possible throughout the table, that is, it should choose each bucket number equally often, or as near to that as possible.

Thus, the nature of the hash function depends on the data type of the key and the number of bytes in the key. Two sample hash functions are given below for strings; they are not perfect, but they give some ideas about what might be done. There is no limit on the number of possibilities.

Hash algorithm 1. This algorithm is appropriate for strings of 1 or 2 characters.

1. Initialize `hash`, an unsigned long, to the first char in the string.
2. Shift hash 7 bits to the left.
3. Add the second character.
4. Return `hash %` table length.

Hash algorithm 2. This algorithm can be used for strings of four or more characters. It uses the first four and last three letters of the string.

1. Set `len` = number of characters in the string, which must be four or greater. (Use `strlen()`.)
2. Set a pointer `p` to the first letter in the string.
3. Set a pointer `q` to the last letter in the string, = `&p[len-1]`.
4. Initialize an unsigned long, `hash` = `*p`;
5. Loop three times:
   (a) left shift `hash` 6 bits.
   (b) exor it with `*q` and decrement `q`.
   (c) left shift `hash` 3 bits.
   (d) exor it with `*(++p)`.
6. Return `hash %` table length.

9.7 Example: Combining Several Data Structures

This sample program is a typing tutor: it provides a list of target words to practice and uses a simple spelling checker to check the accuracy of the user's typing. Its real purpose, however, is to demonstrate the use of several of the data structures presented in this chapter:

- The spelling checker relies on a dictionary of words, which is implemented using a hash table.
- The hash table's buckets are an array of FlexArrays.
- The FlexArrays store pointers to a set of vocabulary words.
- The characters of these words are stored in a StringStore.

9.7.1 The Main Program

```
1  //========================================================================
2  // HashTable, FlexArray, and StringStore Demo Program
3  // A. Fischer, February 7, 2003                          file: Tutor/main.cpp
4  //========================================================================
5  #include "tools.hpp"
6  #include "dictionary.hpp"
7
8  int main ( int argc, char* argv[] )
9  {
10     cout <<"Typing Tutor\n------------\n";
11     if (argc < 2) fatal( "Usage: tutor input_file");
12     Dictionary Dic( argv[1] );  // Create dictionary from words in the input file.
13     cout << "\n\nReady for typing practice? "
14          << "At each prompt, type one of the words shown above.";
15
16     int k, right = 0;           // Prompt for a word 10 times and count correct entries.
17     for (k=0; k<10; ++k) {
18         FlexChars buf;          // Allocate a new flexible char array for the input.
19         cout << "\n> ";
20         buf.inputLine( cin );   // A flexible buffer has no fixed maximum length.
21         if ( Dic.find(buf.message()) ) {    // Is input in dictionary?
22             cout << "\t Correct!";
23             ++right;                        // Keep score.
24         }
25         else {
26             cout <<"\t " << buf <<" is wrong.";
27         }
28     }
29     cout << "\nYou typed "<<right <<" words correctly, out of 10";
30     return 0;
31 }
```

Here is some sample output (several lines have been omitted):

```
Typing Tutor
------------
paper piper pauper pupa puppet pepper

Ready for typing practice? At each prompt, type one of the words shown above.
> paper
    Correct!
...
> puper
    puper is wrong.
> pepper
    Correct!
> payper
    payper is wrong.
You typed 8 words correctly, out of 10
Tutor has exited with status 0.
```

Notes on the main program. This main program is typical for an OO-application: it creates one object (a Dictionary, line 12) and uses that object in a simple way (line 21). Almost all of the work is done within the classes.

The name of the input file is supplied as a command-line argument so that it will be very easy to change the vocabulary. The main program picks up that argument and sends it to the Dictionary constructor (lines 11–12). When dictionary construction is complete, the user is prompted to enter a series of words from the given list, and the tutor checks each one for correctness and counts the correct entries. After ten trials, the typist's score is displayed.

The input buffer used here (line 18) is a flexible array of characters; this means that any input word, no matter how long, can be safely read. To test that property, I ran the program again and entered a string of 142 random characters. The program dutifully read and echoed them all, stated that the spelling was wrong, and went on with the game. Having an input buffer of unlimited length is overkill in this demo program. However,

in a real application, being free from the need to limit the length of an input string can make it much easier for a programmer to achieve bug-free program operation. An added simplification is that a new FlexChars object is created for each input line, making it unnecessary to worry about reinitializing the data structure.

9.7.2 The Dictionary Class

```
32  //=========================================================================
33  // Class declarations for Dictionary.
34  // A. Fischer, April 25, 2001                          file: dictionary.hpp
35  #ifndef DICT
36  #define DICT
37
38  #include "tools.hpp"
39  #include "sstore.hpp"
40  #include "flexChars.hpp"
41  #include "flexString.hpp"
42
43  //=========================================================================
44  class Dictionary {
45    private:
46      StringStore SS;               // Vocabulary is stored in the StringStore.
47      FlexString* HT;               // Vocabulary is accessed using a HashTable.
48      ifstream fin;                 // Stream for reading the vocabulary file.
49      int NumBuk;                   // Number of buckets in the HashTable.
50      int CurrentBuk;               // Bucket currently in use; set by hash.
51      void hash( const char* s );   // Sets CurrentBuk to correct bucket for s.
52      void put( char* s );          // Put new words in the dictionary.
53
54    public:                         //----------------------------------------
55      Dictionary( char* infile, int buckets=101 );
56      ~Dictionary(){ delete[] HT; }
57      bool find( const char* s );
58  };
59  #endif
```

This class implements the dictionary to be used by the typing tutor. Dictionary words are stored in a StringStore (SS) and organized by a hash table (HT) which is implemented as an array of FlexArrays. These powerful data structures are not needed for the little list of words used by the typing tutor demo. However, the hash table and StringStore would be essential tools in a real spelling checker that must handle thousands of words.

Line 104 is commented out, but was very helpful during debugging. You will see similar debugging lines, commented out, in several classes in this program. They have been left in the code as examples of the kind of debugging information that you are likely to need.

The data members of Dictionary All data members are private because all classes should protect data members. They should not be accessed directly by a client program.

Two purposes of the first two data members are obvious: `StringStore SS` and `FlexString* HT` implement the primary data structure of the dictionary. Note that the hash-table array is dynamically allocated (line 101). The third data member is an `ifstream`. It is part of this class because the vocabulary list that is in the input file is only relevant to the dictionary, not to the main application. Declaring an input or output stream within a class is common when the class "owns" the stream. It must be a class member (rather than a local variable in one function) if more than one class function needs to use the stream.

```
60  //=========================================================================
61  // Class implementation for the Dictionary.
62  // A. Fischer, April 25, 2001                          file: dictionary.cpp
63
64  #include "dictionary.hpp"
65  //-------------------------------------------------------------------------
66  void
67  Dictionary::hash( const char* s ){
68      int len = strlen(s);
```

```
69      const char* p = s;
70      const char* q = &p[len-1];
71      unsigned long hashval = *p;
72      for(int k=0; q>=s && k<3; ++k){
73          hashval = hashval << 6 ^ *q--;
74          hashval = hashval << 3 ^ *(++p);
75      }
76      CurrentBuk = hashval % NumBuk;
77  }
78
79  //-------------------------------------------------------------------------
80  void
81  Dictionary::put( char* word ){
82      char* saved;            // Pointer to beginning of word in StringStore.
83      if (!find( word )){
84          saved = SS.put( word, fin.gcount() -1 );
85          HT[CurrentBuk].put( saved );
86      }
87  }
88
89  //-------------------------------------------------------------------------
90  bool
91  Dictionary::find( const char* s ){
92      hash( s );
93      int result = HT[CurrentBuk].find( s );
94      return result >= 0;         // -1 indicates not found.
95  }
96
97  //-------------------------------------------------------------------------
98  Dictionary::Dictionary( char* infile, int buckets ){
99      fin.open( infile );
100     if (!fin)  fatal( "Cannot open %s for input - aborting!!", infile );
101     HT = new FlexString[ NumBuk = buckets ];
102
103     char buf[80];
104     //cerr <<"Reading words from file " <<infile <<".\n";
105     for (;;) {
106         fin >> ws;
107         fin.getline( buf, 80 );
108         if (!fin.good()) break;
109         put( buf );
110         cout << buf <<" ";
111     }
112     if ( !fin.eof() ) fatal( "Read error on tutor file" );
113     fin.close();
114 }
```

Using the state of the object. The last two data members, `Numbuk` and `CurrentBuk` are *state variables*. A state variable is a private data member that describe some essential property of the class or stores the current setting of some internal pointer or index. `Numbuk` is set by the constructor and used to compute every hash index.

In an OO-program, two functions in the same class often use state variables to communicate with each other. In this program, `CurrentBuk` stores the number of the bucket in which the current data item belongs. It is set by hash() and usedlater by find() and put(). This is more efficient and more secure than recalculating the bucket number every time it is needed.

Private function members of Dictionary.

- NumBuk (line 49) is the number of buckets in the hash-table array. The hash() function (lines 51 and 66...77) selects a bucket number (0...NumBuk-1) by making a meaningless computation on the bits of

the data to be stored in the table. These two class members are both private because they relate to the inner workings of the Dictionary class.

- A hashing function is supposed to compute a legal array subscript that has no apparent relationship to its argument. It must also be efficient and repeatable, that is, the same argument must always hash to the same subscript. However, if the hash function is called many times to hash different arguments, it is supposed to distrubute the data evenly among the legal bucket numbers. Bit operations (such as xor and shift) are often used to combine, shift around, and recombine the bits of the argument so that they end up in a seemingly-random (but repeatable) pattern (lines 73 and 74). The mod operator `%` is always used as the last step to scale the final number to the size of the array (line 76).

- The put() function (lines 52 and 80...87) is used by the constructor to build the vocabulary list. It searches the table for a word, then puts the word into the table if it is not there already. This function is private because it is used only by the class constructor; it is not designed to be used by the client program. A more general implementation of a hash table would also provide a public function (with a reasonable interactive interface) that would allow the user to enter new words and store them in the Dictionary by calling this private put() function.

Public function members of Dictionary.

- The constructor. Given the name of a file containing vocabulary words, the Dictionary constructor builds and initializes a dictionary. There are two parameters: the input file name is required, and the length of the hash array is optional. If the optional integer is not supplied by main, in the Dictionary declaration, a default value of 101 will be used. (See Chapter 9, Bells and Whistles, for an explanation of default parameters.)

  This is a long function because it opens (lines 99–100), reads (lines 106–07), processes (lines 108–11), and closes (line 113) the input file. To process one word or phrase from the file, we first test for valid input, then delegate the rest of the job to the `put` function.

- The find() function (lines 57 and 90...95) searches the dictionary for a given word and returns a boolean result. This is a public function becausethe purpose of the class is to allow a client program to find out whether a word is in the dictionary.

  To find out whether or not a word is in the dictionary, we must first select the right bucket then search that bucket. The `hash` function computes the number of the bucket in which the argument string, `s`, should be stored. We then *delegate* the remainder of the task, searching one bucket, to the `Bucket::find` function, and return the result of that function.

- The put() function. It is quite common to name a function `put` if its purpose is to enter a data item into a data structure. Following this naming convention, we would define put() instead of push() for a stack or enqueue() for a queue.

  This function takes care of its own potential problems. Because it is undesirable to have the same item entered twice into a hash table, the first action of `put()` (line 83) is to call `find` to check whether the new word is *already* there. If so, there is no further work to do. If not, two actions are required: to enter the word into the stringstore (line 84) using StringStore::put() , and to enter the resulting pointer into the appropriate bucket of the hash table (line 85) using FlexString:put(). Both actions are delegated to the `put()` functions of the servant classes.

- The destructor. The task of a destructor is to free any and all storage that has been dynamically allocated by class functions. In this class, only the constructor calls new, so the destructor has only one thing to delete. The StringStore was created by declaration as part of the Dictionary, and will be freed automatically when the Dictionary is freed, at the end of `main()`.

  The form of the delete command follows the form of the call on new: Line 101 allocates an array of class objects, so line 56 uses delete[] to deallocate that array; Line 101 stores the allocation pointer in HT, so line 56 deletes HT. Attempting to delete anything else, such as SS or CurrentBuk would be a fatal error.

9.7.3 The FlexArray and StringStore Classes

The code for StringStore was given earlier in this chapter. The code for a flexible array of characters (FlexChars) was also given. We use both classes here without modification.

For this application, we add another class, `FlexString`, a flexible array of C strings. It is like the `FlexChars` except:

- The base type of the flexible array is `char*`, not `char`.
- We added a function to search the array for a given string.
- We added an extension of the subscript operator to permit random access to the array.

The revised and extended class is given below. Later in the term we will see how two powerful techniques, templates and derivation, let us create variations of important data structures without rewriting all fo the code.

The FlexString class declaration. The basic operation of a flexible array was described in Section 8.2 and will not be repeated. However, some new techniques were used here and they deserve some mention. Note the extensive debugging print statements that were needed in the course of writing this code.

- The typedef (line 122). This is a flexible array of strings, or `char*`. The pointer to the head of the array is, therefore, a `char**`. We have defined a typedef name, `handle`, for this type because It is syntactically awkward and confusing to use types that are pointers to pointers. Traditionally, a *handle* is a pointer to a pointer to an object; this term is used frequently in the context of windowing systems.
- The constructor, Line 132, was rewritten to use ctor initializers instead of assignment to initialize the members of a new FlexString. Ctors will be described fully in a future chapter; at this time, you should just be aware that an initializer list (starting with a colon) can follow the parameter list of a constructor function. In the initializer list, member names are given, followed by initial values in parenthess. Following the list is the opening bracket of the body of the constructor function. More code can be given in the body, if needed, or it can be empty, as it is here.
- The find() function. When searching for one data item in an array of data items, you must use a comparison function. The function strcmp() is appropriate for comparing cstrings, so it is used here. The `find` function must compare its argument to elements in the array. Since the kind of comparison to be used depends on the base type of the array, we must either write a new `find` for every array base type or define a `find` that takes a comparison function as a parameter. We chose the first option here.
- The subscript function. Chapter 10 will describe operator extensions. Here, it is enough to note that we can extend the meaning of subscript to access array-like structures that are not just simple arrays. Like the built-in subscript function, the new function extension returns a reference, allowing us to either store data in the array or retrieve it.

```
115  //-------------------------------------------------------------------------
116  // Class declaration for a flexible array of base type T.
117  // A. Fischer, A. Fischer, February 7, 2003            file: flexString.hpp
118  #ifndef FLEXS
119  #define FLEXS
120  #include "tools.hpp"
121  #define STRINGS_START 20    // Default length for initial array.
122  typedef char** handle ;
123
124  class FlexString {
125    private: // ------------------------------------------------------------
126      int Max;               // Current allocation size.
127      int N;                 // Number of array slots that contain data.
128      handle Data;           // Pointer to dynamic array of char*.
129      void grow();           // Double the allocation length.
130
131    public: // -------------------------------------------------------------
```

```
132     FlexString( int ss = STRINGS_START ): Max(ss), N(0), Data(new char* [Max]){}
133     ~FlexString() { if (Data != NULL) delete[] Data; }
134
135     int put( char* data );          // Store data and return the subscript.
136     int length() { return N; }      // Provide read-only access to array length.
137
138     char*& operator[]( int k );     // Extend subscript operator for new class.
139     int find( const char* );        // Return subscript of argument, if found.
140     handle extract();               // Convert a flex-array to a normal array.
141
142     ostream& print(ostream& out) {      // Print the filled part, ignore the rest.
143         for (int k=0; k<N; ++k) out <<Data[k] <<endl;
144         return out;
145     }
146 };
147 inline ostream& operator <<( ostream& out, FlexString F ){ return F.print(out); }
148 #endif


149 //-------------------------------------------------------------------------
150 // Implementation of the FlexString class.
151 // A. Fischer, A. Fischer, February 7, 2003            file: flexString.cpp
152
153 #include "flexString.hpp"
154 int //------------------------ search for the word in the dictionary array.
155 FlexString::find( const char* word) {
156     int k;
157     //cerr <<"    Looking for " <<word <<"  " <<"N is " <<N <<"  ";
158     for (k=0; k<N; ++k) {
159         //cerr <<Data[k] <<"  ";
160         if (strcmp( word, Data[k] ) == 0) break;
161     }
162     return (k < N) ? k : -1;
163 }
164
165 int // --------------------------------- Copy a char* into the FlexString.
166 FlexString::put( char* data ) {
167     if ( N == Max ) grow();    // Create more space if necessary.
168     //cerr <<"Putting " <<data <<" into bucket " <<N <<endl;
169     Data[N] = data;
170     return N++;                // Return subscript at which item was stored.
171 }
172
173 char*& //------------------------------ Access the kth string in the array.
174 FlexString::operator[]( int k ) {
175     if ( k >= N ) fatal("Flex_array bounds error.");
176     return Data[k];               // Return reference to desired array slot.
177 }
178
179 void  // ------------------------------------ Double the allocation length.
180 FlexString::grow() {
181     handle temp = Data;                     // hang onto old data array.
182     Max>0 ? Max*=2 : Max = STRINGS_START;
183     Data = new char*[Max];                  // allocate a bigger one.
184     memcpy(Data, temp, N*sizeof(char*));    // copy info into new array.
185     delete temp;                            // recycle (free) old array.
186                 // but do not free the things that were contained in it.
187 }
```

9.7.4 A Better Way

The FlexChars class and FlexString class are the same except for the type of data item being stored in the array and minor differences in the functions provided by the class. Yet, because of these small differences, we have written and debugged the code twice and used both versions in the same program. There's got to be a better way to accomplish this goal. In fact, there are several C++ tools that, together, provide a better way.

1. If we want to use a standard data structure, and if we want only one version of that data structure, we can define the class with an abstract base type name, like T, and use a `typedef` at the beginning of the program file to map T onto a real type. This method was illustrated by the generic insertion sort in Chapter 2.

2. The C++ standard library supports several basic data structures in the form of *templates*. These templates are abstract code with type parameters, written using techniques very much like those used in the generic insertion sort. These templatea are often referred to as the "Standard Template Library", or STL. `Vector` is the template type corresponding to our flexible arrays. To use a template from the library, you must supply a real type in any the declaration that creates an object of the template class; technically, we say that you `instantiate` the template. The complier will combine your type with the template's abstract code at compilation time, to produce normal compilable code.

   STL is an amazing piece of work. It covers all of the most important data structures, and brings them into a unified interface, so that a program written using one data structure could be changed to use another simply by changing one line of code: the line that instantiates the template. The implementations it provides are efficient and correct, and their performance characteristics are thoroughly documented.

3. You could write your own template type. Templates will be covered in Chapter 13, where we will revisit the flexible array class.

4. You could start with a standard template type or with your own template, and use derivation to add more functions to the class. Such functions might be needed for some base types but not for others. Derivation will be covered in Chapters 12, 14, 15, and 16.