APPENDIX

A The Standard Template Library

The recently adopted C++ Standard requires all implementations to provide a supporting library known as the *Standard Template Library* (known simply as the *STL*). The STL provides a collection of data structures (such as lists, stacks, queues, and priority queues), and algorithms (such as sorting and selection). As its name suggests, the STL makes heavy use of templates, including advanced template features that do not work on many current compilers (and which we have therefore elected not to discuss in this text). As a result, at the time of this writing, there are no completely correct implementations of the STL, although it is certain that correct implementations will appear. It is interesting to examine the STL because it illustrates many of the concepts that have been explored in this text. We will also see that even though the data structures package developed in this text has only basic methods, using it is very similar to using a more robust package, such as the STL.

In this Appendix, we

- describe the organization of the STL, and its integration with the rest of the language
- examine its lists, sets, and maps
- provide a C++ programs that uses the STL

A.1 Introduction

The STL contains implementations of some of the data structures that have been described in this text. Specifically, there is a doubly-linked list class, with an associated iterator, priority queues, and data structures that make use of balanced search trees. As expected, the functionality of these classes is somewhat different than the classes written in this text; however the basic concepts, algorithms, and running times are the same. The STL does not provide a hash table data structure or a union/find data structure. There is a binary search algorithm and a quicksort algorithm.

Because the STL is part of the C++ library, it is likely to undergo extensive testing and optimization, and have survived use by legions of programmers around the world. Thus, in general, it will be preferable to use it, rather than provide an alternate implementation.

Complete coverage of the STL fills a textbook. In this appendix, we restrict our attention to a small subset that includes the basics of the STL.

A.2 Basic STL Concepts

This section describes the basics of the STL, including the new header files, the *`using` directive*, containers and iterators, pairs, and function objects.

A.2.1 Header Files and The `using` Directive

Historically, the names of library header files have ended with the `.h` suffix. The new standard mandates that these names are now suffix-free. Thus, the standard I/O header file is now `iostream`, instead of `iostream.h`. Many implementations will continue to provide an `iostream.h` header file. However, this file may not be compatible with the STL version. In Visual C++ 5.0, for instance, you cannot use `iostream.h` if you use any of the STL header files. Some of the other header files are `fstream`, `sstream`, `vector`, `list`, `deque`, `set`, and `map`.

The newly adopted standard also adds a new feature called the *namespace*. Although namespaces are important in their own right, we do not discuss their use here. It is important to know, however, that the entire STL is defined in the `std` namespace. To access the STL as if it were in the global namespace we provide a *`using` directive*, which in this case is:

```
using namespace std;
```

Although there are other alternatives, which can be found in recent C++ books, this is the simplest. Figure A.1 illustrates the new `iostream` header file and the `using` directive.

A.2.2 Containers

A *container* represents a group of objects, known as its elements. Some implementations, such as vectors and lists, are unordered; others, such as sets and maps are ordered. Some implementations allow duplicates, others do not. All containers support the following operations.

`bool empty( ) const`

Returns `true` if the container contains no elements; `false` otherwise.

`iterator begin( ) const`

Returns an `iterator` that can be used to begin traversing all locations in the container.

`iterator end( ) const`

Returns an `iterator` that represents the "end marker," or a position past the last element in the container.

`int size( ) const`

Returns the number of elements in the container.

The most interesting of these methods are those that return an `iterator`. The operations that can be performed by an `iterator` are described in Section A.2.3.

A.2.3 `iterator`

There are actually many types of `iterator`s. However, we can always count on the following operations being available for any iterator type:

`itr++`

Advances the iterator `itr` to the next location. Both the prefix and postfix forms are allowable, but the precise return type (whether it is a constant reference or a reference) can depend on the type of iterator.

`*itr`

Returns a reference to the object stored at iterator `itr`'s location. The reference returned may or may not be modifiable, depending on the type of iterator. For instance, the `const_iterator`, which is used to traverse `const` containers, has an `operator*` that returns a `const` reference, thus disallowing `*itr` being on the left-hand side of an assignment.

```
#include <iostream>
using namespace std;

int main( )
{
    cout << "First program" << endl;
    return 0;
}
```

Figure A.1 First program using new STL

```
// Print the contents of Container c
template <class Container>
void printCollection( const Container & c )
{
    Container::const_iterator itr
    for( itr = c.begin( ); itr != c.end( ); itr++ )
        cout << *itr << '\n';
}
```

Figure A.2 Print the contents of any `Container`

`itr1==itr2`

Returns `true` if iterators `itr1` and `itr2` refer to the same location; `false` otherwise.

`itr1!=itr2`

Returns `true` if iterators `itr1` and `itr2` refer to a different location; `false` otherwise.

Each container defines several iterators. For instance, a `list<int>` defines `list<int>::iterator` and `list<int>::const_iterator`. (There are also reverse iterators, that we do not discuss.) The `const_iterator` must be used instead of an `iterator` if the container is non-modifiable.

As an example, the routine in Figure A.2 prints each element in any container, provided that the element has `operator<<` defined for it. If the container is an ordered set, its elements are output in sorted order.

A.2.4 Pairs

Often it is necessary to store a pair of objects in a single entity. This is useful for returning two things simultaneously. It is also useful for the `map` class, discussed in Section A.5. The STL defines a template `pair` class with the following semantics:

```
template <class Object1, class Object2>
class Pair
{
  public:
    Object1 first;
    Object2 second;
};
```

A.2.5 Function Objects

Container algorithms that require an ordering property generally use a default order (typically the `less` function, implemented as a call to the object's `operator<`). The algorithms can generally provide a function that specifies a different ordering property. This is most useful when the natural ordering is not exactly what is needed. For instance, we may want to sort a `vector` of strings, but ignore case distinctions. Or for a simpler example, we may want to sort the strings by their length.

An example is shown in Figure A.3; the function `Comp` compares strings by length; this function is passed as the optional third parameter to `sort` in the form of an object. A *function object* defines an implementation for its `operator()`, which is the function call operator. We then pass an instance of the function object as the third parameter to `sort`.

Although this function object contains no data members and no constructors, more general function objects are possible. The only requirement is that `operator()` must be defined. The STL provides numerous template function objects including `less` (the default for many container algorithms) and `greater`.

```
class Comp
{
  public:
    bool operator( )( const string & lhs,
                      const string & rhs ) const
      { return lhs.length( ) < rhs.length( ); }
};

void sortListOfStringsByLength( vector<string> & array )
{
    sort( array.begin( ), array.end( ), Comp( ) );
}
```

Figure A.3 A sorting algorithm using a function object

A.3 Unordered Sequences: `vector` and `list`

Both a `vector` and `list` can be used to implement an unordered container (also known as a sequence). The user has precise control over where in the sequence each element is inserted. The user can access elements by their position in the sequence, and search for elements in the sequence. However, depending on the particular operation, only one of the `vector` or `list` might be efficient.

A.3.1 `vector` vs. `list`

The STL provides three sequence implementations, but only two are generally used: an array-based version and a doubly linked-list based version. The array-based version can be appropriate if insertions are performed only at the high end of the array, for the reasons discussed in Chapter 3. The STL doubles the array if an insertion at the high-end would exceed capacity. Although this gives good Big-Oh performance, for large objects that are expensive to construct, a list version would be preferable in order to minimize calls to the constructors.

Insertions and deletions toward the middle of the sequence are inefficient in the `vector`; on the other hand, direct access by the index is impossible in a `list`. If indexing is not needed, the `list` can always be safely used. The `vector` may still be a better choice if insertions occur only at the end and the objects being inserted are not overly expensive to construct. Some of the additional operations on sequences are:

`void push_back( const Object & element )`

Appends `element` at the end of this sequence.

`void push_front( const Object & element )`

Prepends `element` to the front of this sequence. Not available for `vector`, because it is too inefficient. However a `deque` is available that is like a `vector`, but supports double-ended access.

`Object & front( ) const`

Returns the first element in this sequence.

`Object & back( ) const`

Returns the last element in this sequence.

`void pop_front( )`

Removes the first element from this sequence. Available only for `list` and `deque`.

`void pop_last( )`

Removes the last element from this sequence.

`iterator insert( iterator pos, const Object & obj )`

Inserts `obj` prior to the element in the position referred to by `pos`. This operation takes constant time for a `list`, but takes time proportional to the distance from `pos` to the end of the sequence for a `vector`. Returns the position of the newly inserted item.

`void erase( iterator pos )`

Removes the object at the position referred to by `pos`. Elements in the sequence are logically moved as required. This operation is constant time for a `list`, but takes time proportional to the distance from `pos` to the end of the sequence for a `vector`. Returns the position of the newly inserted item.

A.3.2 Stacks and Queues

The STL provides a `stack` and `queue` class, but these simply use a sequence container (`list`, `vector`, or `deque`), calling the appropriate functions. The queue does not even use standard names such as `enqueue` and `dequeue`. Thus there's no compelling reason not to use the sequence containers directly.

A.4 Sets

The `set` is an ordered container. It allows no duplicates.[1] The underlying implementation is a balanced search tree. In addition to the usual `begin`, `end`, `size`, and `empty`, the `set` provides:

`pair<iterator,bool> insert( const Object & element )`

Adds `element` to the `set` if it is not already present. The `bool` component of the return value is `true` if the set did not already contain `element`; otherwise it is `false`. The `iterator` component of the return value is the location of `element` in the set.

`iterator find( const Object & element ) const`

Returns an `iterator` containing the location of `element` in the set, or `end()` if `element` is not in the set.

`int erase( const Object & element )`

Removes `element` from the set if is present. Returns the number of elements removed (thus, either 0 or 1).

By default, ordering uses the `less<Object>` function object, which itself is implemented by calling `operator<` for the `Object`. An alternate ordering can be specified by constructing the `set` with a function object.

A.5 Maps

A `map` is used to store a collection of ordered entries that consists of *keys* and their *values*. The `map` maps keys to values. Keys must be unique, but several keys can map to the same values.[2] Thus values need not be unique. The `map` uses a balanced search tree to obtain logarithmic search times.

The map behaves like a `set` instantiated with a `pair`, whose comparison function refers only to the key. Thus it supports `begin`, `end`, `size`, and `empty`, but the underlying iterator is a key,value pair. In other words, for an iter-

1. The `multiset` allows duplicates, but we do not discuss the `multiset` here.
2. The `multimap` allows duplicate keys, but we do not discuss the `multimap` here.

ator `itr`, `*itr` is of type `pair<KeyType,ValueType>`. The `map` also supports `insert`, `find`, and `erase`. For `insert`, one must provide a `pair<KeyType,ValueType>` object. Although `find` only requires a key, the iterator it returns references a `pair`. Using only these operations is hardly worthwhile, because the syntactic baggage can be excessive.

Fortunately the map has an important extra operation. The array-indexing operation is overloaded for maps:

`ValueType & operator[]( const KeyType & key )`
`const ValueType & operator[]( const KeyType & key ) const`

Returns the value to which this `map` maps `key`. If `key` is not mapped then `key` becomes mapped to a default `ValueType` generated by applying a zero-parameter constructor.

This type of syntax is sometimes known as an *associative array*. Although we'll see an example of the `map` shortly, it is worth illustrating with a few lines of code. In Figure A.4, `people` maps a `string` to an `int`. So `"Tim"` is initially 3, and then 5, which is output by the first print statement. `"Bob"` is not in the `map` prior to the print statement, but the call to `operator[]` puts it in the `map` with a default value of 0. Thus 0 is (perhaps unintentionally) output by the second print statement. To know if `"Bob"` was in the `map`, we would have needed to call `find` first, and check to see if the returned iterator was equal to `end()`. Once we call `find`, since we have an iterator `itr`, to `find` the value, we should use `itr->second`, to avoid a second search.

A.6 Example: Generating a Concordance

A concordance of a file is a listing that contains all the words in a file, with the line number on which the word occurs. Using the STL, we can write a program that produces a concordance. We assume that a word is any sequence of consecutive non-white space characters.

```
#include <map>
using namespace std;

int main( )
{
    map<string,int> people;
    people["Tim"] = 3; people["Tim"] = 5;
    cout << "Tim's value is " << people["Tim"] << endl;
    cout << "Bob's value is " << people["Bob"] << endl;
    return 0;
}
```

Figure A.4 Illustration of the `map`. Tim's value is 5. Bob's value is 0.

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <map>
#include <string>
#include <vector>
using namespace std;

ostream & operator<<( ostream & out,
                      const pair<string, vector<int> > & rhs )
{
    out << rhs.first << ": " << '\t' << rhs.second[ 0 ];
    for( int i = 1; i < rhs.second.size( ); i++ )
        out << ", " << rhs.second[ i ];
    return out;
}

int main( int argc, char *argv[ ] )
{
    if( argc != 2 )
    {
        cerr << "Usage: " << argv[ 0 ] << " filename" << endl;
        return 1;
    }

    ifstream inFile( argv[ 1 ] );
    if( !inFile )
    {
        cerr << "Cannot open " << argv[ 1 ] << endl;
        return 1;
    }

    typedef map<string, vector<int> > wordmap;
    wordmap concordance;
    string  oneLine;

        // Read the words; add them to wordmap
    for( int lineNum = 1; getline( inFile, oneLine ); lineNum++ )
    {
        istringstream st( oneLine );
        string word;
        while( st >> word )
            concordance[ word ].push_back( lineNum );
    }

    wordmap::iterator itr;
    for( itr = concordance.begin( ); itr != concordance.end( ); itr++ )
        cout << *itr << endl;
    return 0;
}
```

Figure A.5 Concordance program using the STL

The basic idea is to use a `map`, to map words to a list of lines on which the word occurs. Thus each key is word, and its value is a list of line numbers. When we see a word, we check to see if it is already in the map. If it is, then we simply add the current line number to the list of lines that corresponds to the word. If it is not, we add to the map the word along with a list containing the current line number. After we have read all of the words, we can iterate through the map. This generates the map entries in key-sorted order, so the words will appear in sorted order. For each map entry, we output the word, and then we go through the linked list of line numbers, and output them

A.6.1 STL Version

The code that uses the STL is shown in Figure A.5. We open a file and create a `map`. We can use either a `vector` or `list` to store the line numbers, since both support efficient `push_back` operations. In the `for` loop, we repeatedly read one line at a time, maintaining the current line number. The `istringstream` is used to extract white-space-delimited tokens from the line (it has the same look and feel as any other stream). The line number is then added to the entry corresponding to `word` in the concordance `map`. (When a word is seen for the first time, the expression `concordance[word]` inserts the pair consisting of `word` and a default `vector` into the map. Thus the subsequent `push_back` is safe.) At the end of the loop, we use an iterator to go through the map, and print out each map entry.

The overloaded `operator<<` function accepts a pair; the word is stored in the `first` data member and the `vector` of line numbers in the `second` data member. The code treats the first line number as a special case (it is not preceded by a comma, but is preceded by a tab); it is otherwise similar to the code in Figure A.2. Note that `operator<<` assumes that the list of line numbers is not empty, which is guaranteed by the rest of the code.

As I write this, the program works only on Visual C++ 5.0. By using a different string stream class, we can get it to work on g++ 2.8.1.

A.7 Other STL Features

The STL is a powerful library that can be very useful for many applications. We have discussed only the bare bones basics. The STL contains many other interesting constructs that we do not discuss.