Chapter 19: The Template Library

From a review of *Effective STL : 50 Specific Ways to Improve Your Use of the Standard Template Library* by ScottMeyers:

> It's hard to overestimate the importance of the Standard Template Library: the STL can simplify life for any C++ programmer who knows how to use it well. Ay, there's the rub.

19.1 The Standard Template Library

STL was designed with extreme care so that it is complete and portable and as safe as possible within the context of standard C++. Among the design goals were:

- To provide standardized and efficient implementations of common data structures as templates, and of algorithms that operate on these structures.
- To produce efficient code. Instantiation of the generic container class is done at compile-time, producing code that is both correct and efficient at run time. This contrasts with derivation and polymorphism which can be used to achieve the same ends but are much less efficient at run time.
- To unify array and linked list concepts, terminology, and interface syntax. Code can be written and partially debugged before making a commitment to one kind of implementation or to another. This permits code to be designed and built in a truly top-down manner,

There are three major kinds of components in the STL:

- Containers manage a set of storage objects (list, tree, hashtable, etc). Twelve basic kinds are defined, and each kind has a corresponding allocator that manages storage for it.
- Iterators are pointer-like objects that provide a way to traverse through a container.
- Algorithms are computational procedures (sort, set_union, make_heap, etc.) that use iterators to act on containers and are needed in a broad range of applications. Function objects encapsulate a function in an object and can be used in algorithms instead of function pointers.

In addition to these components, STL has several kinds of objects that support ones including pairs (key–value pairs, for the associative containers), allocators (to support dynamic allocation and deallocation) and function-objects (to "wrap" a function in an object).

19.2 Iterators

An iterator provides a general way to access the objects in a container. The underlying value can be mutable or constant. This concept unifies and replaces both subscripts and pointers. Exceptional values are also defined:

- Past-the-end values (not dereferenceable)
- Singular values (garbage).

There are five types of iterators, related like this: The last two types are called "reverse iterators": they are able to traverse a container backwards, from end to beginning.

- Input iterators A input iterator class must have a constructor and support the operators `->`, `++`, `*`, `==`, and `!=`, and the `*` operator cannot be used as an l-value in an assignment.

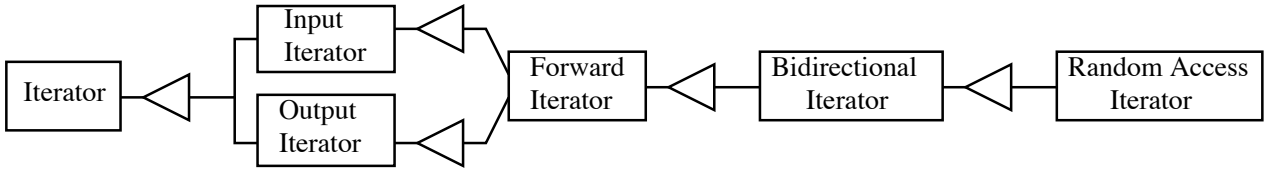

Figure 19.1: Iterator classes form a type hierarchy:

- Output iterators support writing values to a container (using `=` but not reading from it. Restriction: assignment through the same iterator happens only once.
- Forward iterators can traverse the container from beginning to end (using ++)
- Bidirectional iterators can go back and forth in the container; they support the operator `--` in addition to the basic list.
- Random-access iterators must support these operators in addition to the basic list: `[], --, <. >, <=, >=, -=, and +=`. Further, the * operator must support assignment..

The purpose of iterators is clear: to replace both subscripts and pointers, and allow the traversal of all elements in an container in a uniform syntax that does not depend on the implementation of the container or on its content-type. The different types of iterators allow a programmer to select the kind of traversal that is needed and the restrictions (read-only or write-only) to be placed on access. The examples in Section 13.8 will make all this clearer.

19.3 Containers

The definition of each container class consists of template code, of course, but that code is not part of the standard. Instead, the standard gives a complete definition of the functional properties and time/space requirements that characterize the container. Two groups of containers are supported: sequence containers (lists, vectors, queues, etc.) and sorted associative containers (maps, sets, etc). The intention is that a programmer will select a class based on the functions it supports and its performance characteristics. Although natural implementations of each container are suggested, the actual implementations are not standardized: any semantics that is operationally equivalent to the model code is permitted. Big-Oh notation is used to describe performance characteristics. In the following descriptions, an algorithm that is defined as time O(n), is no worse than O(n) but may be better.

The basic building blocks are precisely organized and, within a group, interchangeable.

Member operations. Some member functions are defined for all containers. These include:

- Constructors: A null constructor, a constructor with one parameter of the container type, and a copy constructor. The latter two constructors operate in linear time.
- Destructor: It will be applied to every element of the container and all memory will be returned. Takes linear time.
- Traversal initialization: begin(), end(), rbegin(), rend(). These mark beginning and ending points for a traversal or reverse-traversal.
- The object's state: size() – current fill level, max_size() – allocation size, empty() – true or false.
- Assignment: `=` Assign one container to another. Linear time.
- Equality: `a == b, a != b` – Returns true or false. Two containers are equal when the sequenes of elements in a and b are elementwise equal (using the definition of operator== on the element type. Otherwise they are not equal Both take linear time.
- Order: `<, <=, >, >=` Lexicographic comparisons; linear time.
- Misc: a.swap(b) – swaps two conainers of the same type. Constant time.

Sequence Containers

These classes all have the following requirements:

- Constructor with two parameters, int n and base-type element t; Construct a sequence with n copies of t.
- Constructor with two forward-iterator parameters, j and k. Construct a sequence equal to the contents of the range [j, k).
- Traversal initialization: begin(), end(), rbegin(), rend(). These mark beginning and ending points for a traversal or reverse-traversal.
- The object's state: size() – current fill level, max_size() – allocation size, empty() – true or false.
- Assignment: `=` Assign one container to another. Linear time.
- Equality: `a == b, a != b` – Returns true or false. Two containers are equal when the sequenes of elements in a and b are elementwise equal (using the definition of operator== on the element type. Otherwise they are not equal Both take linear time.
- Order: `<, <=, >, >=` Lexicographic comparisons; linear time.
- Misc: a.swap(b) – swaps two conainers of the same type. Constant time.

Sorted Associative Containers

All associatiative containers have two parameters: a type `Key` and an ordering function `comp`, called the "comparison object" of the container, that implements the idea of `<=`. Two keys, `k1, k2` are *equivalent* if `comp(k1, k2` and `comp(k2, k2` both return false. These classes all define the following functions:

- Constructors that include a comparison-object as a parameter.
- Selectors that return the key_type, comparison object-type, and comparison objects.
- Insertion and deletion: four insertion functions with different parameters named `insert()` and `uniq_insert()`. Three `erase()` functions.
- Three iterator functions: `lower_bound(), upper_bound(), and equal_range()`
- Searching: `find(k)`, which returns a pointer to the element whose key is k, (or `end()` if such an element does not exist), and `count(k)` which returns the number of elements whose keys equal `k`.

19.3.1 String

```
1  // STL string example. Joseph Parker, modified by A. Fischer, March 26, 2003
2  #include <string>                              // Header file for STL strings
3  #include <iostream>
4  using namespace std;
5
6  int main( void )
7  {
8      string str1 = "This is string number one.";   // Allocate and initialize.
9      // string size
10     cout << "String str1 is: \"" << str1.c_str() <<"\". "
11         <<" Its length is: " << str1.size() << "\n\n";
12
13     cout << "Get a substring of six letters starting at subscript 8: ";
14     string str2 = str1.substr(8,6);
15     cout << str2.c_str() << endl;
16
17     // search first string for last instance of the letter 'e'
18     unsigned idx = str1.find_last_of("e");
19     if (idx != std::string::npos)
20         cout << "The last instance of the char 'e' in string str1 is at pos "
21             << idx <<endl;
22     else cout << "No char 'e' found in string str1" <<endl;
```

```
23
24      // search second string for first instance of the letter 'x'.
25      idx = str1.find_first_of("x");
26      if (idx != std::string::npos)
27          cout << "The first instance of the char 's' in string str2 is at pos "
28               << idx <<"\n\n";
29      else cout << "No char 'x' found in string str2\n\n";
30
31      cout << "Now replace \"string\" with \"xxxyyyxxx\".\n";
32      idx = str1.find("string");
33      if (idx != std::string::npos)
34          str1.replace(idx, string("string").length(), "xxxyyyxxx");
35      cout << "str1 with replacement is \"> " << str1.c_str() << "\"\n";
36      return 0;
37  }
```

The output:

```
String str1 is: "This is string number one.".  Its length is: 26

Get a substring of six letters starting at subscript 8: string
The last instance of the char 'e' in string str1 is at pos 24
No char 'x' found in string str2

Now replace "string" with "xxxyyyxxx".
str1 with replacement is "> This is xxxyyyxxx number one."

StringDemo has exited with status 0.
```

Please note these things:

- Line 2: the header function needed for this class.
- Line 10: how to get a C-style string out of a C++ string, and print it.
- Line 11: how to get the length of the string.
- Line 14: creating a new string from a substring of another.
- Lines 18 and 25: searching a string for a letter (first and last occurrenctss).
- Line 34: replace a substring with another string. Note that the old and new substrings do not need to be the same length.

19.3.2 Vector

```
40  // STL_Vector_Example.cpp by Joseph Parker, modified by A. Fischer, March 2003
41  #include <iostream>
42  #include <vector>
43  #include <algorithm>
44
45  using namespace std;
46
47  //---------------------------------------------------------------------
48  void print(vector<int>& v)      // print out the elements of the vector
49  {
50      int idx = 0;
51      int count = v.size();
52      for ( ; idx < count; idx++)
53          cout << "Element " << idx << " = " << v.at(idx) << endl;
54  }
55
56  //---------------------------------------------------------------------
57  int main( void )
58  {
59      vector<int> int_vector;         // create a vector of int's
```

```
60
61      // insert some numbers in random order
62      int_vector.push_back(11);
63      int_vector.push_back(82);
64      int_vector.push_back(24);
65      int_vector.push_back(56);
66      int_vector.push_back(6);
67
68      cout << "Before sorting: " <<endl;
69      print(int_vector);                          // print vector elements
70      sort(int_vector.begin(), int_vector.end()); // sort vector elements
71      cout << "\nAfter sorting: " <<endl;
72      print(int_vector);                          // print elements again
73
74      // search the vector for the number 3
75      int val = 3;
76      vector<int>::iterator pos;
77      pos = find(int_vector.begin(), int_vector.end(), val);
78      if (pos == int_vector.end())
79          cout << "\nThe value " << val << " was not found" << endl;
80
81      // print the first element
82      cout << "First element in vector is " << int_vector.front() << endl;
83
84      // remove last element
85      cout << "\nNow remove last element and element=24 "<< endl;
86      int_vector.pop_back();
87
88      // remove an element from the middle
89      val = 24;
90      pos = find(int_vector.begin(), int_vector.end(), val);
91      if (pos != int_vector.end())
92          int_vector.erase(pos);
93
94      // print vector elements
95      print(int_vector);
96      return 0;
97  }
```

The output:

```
Element 0 = 11
Element 1 = 82
Element 2 = 24
Element 3 = 56
Element 4 = 6

Element 0 = 6
Element 1 = 11
Element 2 = 24
Element 3 = 56
Element 4 = 82

The value 3 was not found
First element in vector is 6
Now remove last element and element=24
Element 0 = 6
Element 1 = 11
Element 2 = 56
```

Please note: A STL vector is a generalization of a FlexArray. You might want to use it because It is not as restricted and presents the same interface as the other STL sequence container classes. The FlexArray, however, is simpler and easier to use for those things it does implement.

- Line 42: the header function needed for this class.
- Line 59: we construct a vector, given the type of element to store within it. Initially this vector is empty.

- Lines 62–66: se put five elements into the vector (at the end).
- Line 69 and 72: we print the vector before and after sorting.
- Line 70: `begin()` and `end()` are functions that are defined on all containers. They return iterators associated with the first and last elements in the container. (It appears that `end()` is actually a pointer to the first array slot past the end of the vector.)
- Line 70: sort is one of the algorithms supported by vector. The arguments to sort are two iterators: one for the beginning and the other for the end of the portion of the vector to be sorted.
- Line 76: we declare an iterator variable of the right kind for vector and use it on the next line to store the position at which a specific element is found.
- Lines 77 and 90: the `find()` function searches part of the vector (specified by two iterators) for a key value (the third argument).
- Line 78 tests for the value `end()`, which is a unique value returned by the find function to signal failure to find the key value.
- Line 82: get the first element in the vector but do not erase it from the vector.
- Line 86: remove the last element from the vector: very efficient.
- Line 92: remove an element from the middle of a vector, using an iterator: not as efficient as `pop()`.

19.3.3 Map

A map is a collection of key/value pairs. When a value is stored into a map, it is stored using one of its fields, called the key field. To retrieve that value, you use the key field to locate it. As STL map can hold only unique keys; if more than one item with the same key can exist, you must use a multimap instead.

This class could be implemented as a hash table or a balanced search tree. Either one would make sense and serve the purpose. We do not actually know which is used, since that is not dictated by the standard. The standard guarantees performance properties, but not implementation. (One might be able to deduce the implementation from the performance properties, though.)

```
100  // STL_Map_Example.cpp by Joseph Parker, modified by A. Fischer, March 2003
101  #include <map>
102  #include <iostream>
103  #include <string>
104  using namespace std;
105
106  int main( void )
107  {
108      // create a map
109      map<int, string> myMap;
110      map<int, string>::iterator it;
111
112      // insert several elements into the map
113      myMap[1] = "Andrea";
114      myMap.insert(pair<int, string>(2, "Barbara"));
115
116      // print all elements
117      for (it = myMap.begin(); it != myMap.end(); ++it)
118          cout << "Key = " << it->first
119              << ", Value = " << it->second
120              << endl;
121
122      // try some operations
123      it = myMap.find(2);
124      if (it == myMap.end()) cout << "\nKey value 2 not found" << endl;
125      else cout <<"\nValue for key 2 = " << it->second << endl;
126
127      it = myMap.find(3);
128      if (it == myMap.end()) cout << "Value for key 3 not found\n";
```

```
129
130       // get # of elements in map
131       cout << "\nThe number of elements in myMap is " << myMap.size() << endl;
132       cout << "Now erase one element from map.\n";
133       myMap.erase(2);
134       cout << "The number of elements in myMap is " << myMap.size() << endl;
135       return 0;
136   }
```

The output:

```
Key = 1, Value = Andrea
Key = 2, Value = Barbara

Value for key 2 = Barbara
Value for key 3 not found

The number of elements in myMap is 2
Now erase one element from map.
The number of elements in myMap is 1

MapDemo has exited with status 0.
```

Please note:

- Line 101: the header function needed for this class.
- Lines 109 and 110 create a map object and an appropriate iterator. Note that both require two parameters: the type of the key field and the type items stored in the container.
- Lines 113 and 114 insert two pairs into the map. Note that the second pair is constructed within the argument list of the insert function.
- Lines 116–120 iterate through the container and print each element. Note that each pair has two members, named first and second, and that these members are used to access both the key value and the data.
- Lines123 and 127 call `map::find()`, supplying the key. Compare this to to the call on `vector::find()` in line 77: the required parameters are different but both return an iterator that points to the found item.
- Lines 124 and 128 test for success of the find operation by comparing the result to `myMap.end()`. This is exactly like the test on line 78 in the vector example.
- Lines 131 and 134 get the number of pairs stored in the map.
- Line 133 removes a specific pair from the map, given its key value.

Conclusion The three STL classes introduced here are among the simplest and most useful. But these examples are "only the tip of the iceberg"; the capabilities of these and other STL classes and algorithms go far beyond what is shown here. The three examples are only a starting point from which the student can continue to learn and master this important aspect of modern programming practice.[1] [2]

[1] The Schildt textbook contains detailed and up-to-date material covering all of the standard templates.
[2] I am also using the first STL book – Musser and Saini, *STL Tutorial and Reference Guide*, and I find it quite readable.