C++ Standard Libraries: An Introduction

ANDREW L. HAZEL
Department of Mathematics, University of Manchester
Oxford Road, Manchester, M13 9PL, UK

Contents

1 Complex Numbers

The library `<complex>` contains the definition of a template-based complex number class, with corresponding definitions of the basic arithmetic operators `+, -, /, *`. The functions `abs(complex<T> &)` and `arg(complex<T> &)` return the absolute value and argument of the complex number and the hyperbolic, logarithmic and circular functions have all been overloaded to handle complex numbers.

The code segment below shows typical use of this class

```
#include<iostream>
#include<complex>

//complex actually lives in the std namespace
using namespace std;
int main()
{
 //Declare two complex numbers 1+i and 1 -i
 complex<double> A(1,1), B(1,-1);
 //Declare a third complex number
 complex<double> C;
 //C is A divided by B (overloaded division operator here)
 C = A/B;

 //This will produce the output i (0,1)
 cout << C << endl;

 //Output the absolute value (1) and argument (pi/2)
 cout << abs(C) << " " << arg(C) << endl;

 //Logarithm of complex number
 cout << log(B) << endl;
}
```

The output from the code is:

```
(0,1)
1 1.5708
(0.346574,-0.785398)
```


2 Standard Containers

A container is an object whose main purpose is to hold other objects, examples include vectors, sets and lists. Objects can be inserted into and removed from containers, and the objects in a container can be sorted and copied into other containers.

The standard containers have different header files, usually the name of the container, and the most useful are shown in Table 1:

Type	Example	Header
`vector<T>`	a variable-sized 1D array	`<vector>`
`list<T>`	a doubly-linked list	`<list>`
`set<T>`	a set	`<set>`
`map<key,val>`	an associative array	`<map>`
`stack<T>`	a stack	`<stack>`
`queue<T>`	queue	`<queue>`

Table 1: Some of the standard container types in C++.

The standard containers are designed to be as similar as possible. For example, every container has a member function `size()` that returns the number of objects in the container. In the following sections, we shall examine some of the standard containers in more detail.

2.1 Vector

A vector, `v`, contains an array of `N` objects indexed from `0` to `N-1` and provides efficient, direct access to the `i`th element via subscripting, `v[i]`, or (with range checking) `v.at(i)`. A vector can be initialised in many different ways:

```
vector<double> vec;      //Empty vector
vector<double> vec2(50); //A vector of 50 doubles
vector<int> vec3(10,5);  //A vector of 10 integers, each with the
                         //value 5
```

The function `resize()` changes the size of a vector, which is therefore a dynamic data structure — it can change in size during the execution of the program. A bonus of using the vector array, rather than writing your own dynamic arrays, is that memory allocation and de-allocation is handled by the class.

```
int n=15; //Integer
vector<double> vec; //Empty vector
//Resize the vector to have n entries
vec.resize(n);
```

2.1.1 Stack Operations

Vectors also support stack operations, such as `push_back()`, which adds an entry at the end, `pop_back()`, which removes the last entry, `front()`, which returns the value of the first entry and `back()`, which returns the value of the last entry.

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

int main()
 {
  //Declare a vector of integers
  vector<int> vec;
  //Loop over the numbers 1 to 10 and add to the vector
  for(unsigned int i=1;i<=10;i++) {vec.push_back(i);}

  //Print out the length of the vector
  cout << vec.size() << endl;

  //Print out the first and last entries of the vector
  cout << vec.front() << " " << vec.back() << endl;

  //Remove the final two entries
  vec.pop_back(); vec.pop_back();

  //Now print out the last entry
  cout << vec.back() << endl;
```



```
 }
```

The output from the above program is:

```
10
1 10
8
```

2.1.2 Iterators

Every standard container provides **iterators** that can be used to step through the objects held in the container. An iterator is merely an object that is used to refer to the objects in the container in some particular order. It can be regarded as a pointer to the objects in the container, but its exact type depends upon the implementation and the container.

For a standard container, `C`, the iterator is always called `C::iterator` and the functions `begin()` and `end()` return iterators to the first object in the container and the last+1 object, respectively, and allow forward traversal of the objects in the container. In order to access the object at the position of the iterator the syntax is like that of pointers.

```
 vector<int> vec(10,5); //An array of 10 integers, all 5
//Print out the value of the first entry, using the iterator
cout << *vec.begin();
```

For a standard container, `C`, there is also a reverse iterator, `C::reverse_iterator` and the functions `rbegin()` returns a reverse iterator to the end of the array and `rend()` returns a reverse iterator to one before the start of the array. These allow reverse traversal of the objects in the container.

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

int main()
 {
  //Declare a vector of ten integers
  vector<int> vec(10);
```

```
  //Fill the vector
  for(unsigned int i=0;i<10;i++) {vec[i] = 2*i;}

  //Use iterators to traverse the vector forwards
  cout << "Traversing Forwards" << endl;
  for(vector<int>::iterator p = vec.begin(); p != vec.end(); p++)
   {
    cout << *p << " ";
   }

  //Print a newline
  cout << endl;

  //Use reverse iterators to traverse the vector backwards
  cout << "Traversing Backwards" << endl;
  for(vector<int>::reverse_iterator p=vec.rbegin(); p!=vec.rend(); p++)
  {
   cout << *p << " ";
  }
}
```

The above code fragment illustrates the use of iterators for vectors and gives the following output

```
Traversing Forwards
0 2 4 6 8 10 12 14 16 18
Traversing Backwards
18 16 14 12 10 8 6 4 2 0
```

In fact, entries in a vector may be traversed by direct access (using subscripts *e. g.* `vec[i]`, but subscripts are not available in all containers. Using iterators is a *generic* way of traversing a standard container and should be used in standard algorithms.

2.1.3 List Operations

Vectors also support list-type operations `insert()` and `erase()` to add and delete elements at given positions. These operations are **not** efficient, but can be very convenient and their use is illustrated in the following program


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

int main()
{
 //Declare a vector of ten integers
 vector<int> vec(10);
 //Fill the vector
 for(unsigned int i=0;i<10;i++) {vec[i] = 2*i;}

 //Insert the number 100 at the beginning of the vector
 vec.insert(vec.begin(),100);

 //Insert two copies of 15 before the last entry in the vector
 vec.insert(vec.end()-2,2,15);

 //Check the size of the vector --- it should be 13
 cout << vec.size();
 //Output the first entry
 cout << " " << vec.front() << endl

 //Now delete the 5th entry of the array
 vec.erase(vec.begin()+4);
 //Delete the last two entries in the array
 vec.erase(vec.end()-2,vec.end());

 //Check the size of the vector --- it should be 10
 cout << vec.size() << endl;
 //Output the last entry
 cout << " " << vec.back() << endl
}
```

which produces the following output

```
 13 100
 10 15
```

2.2 List

A list is designed for efficient insertion and deletion of objects at given positions. It does not have a subscript `[]` operator, but otherwise has all the functionality of a vector. In addition, a list contains the `push_front()` and `pop_front()` functions for adding or removing entries from the start of the list efficiently. A list also contains the useful `remove()` function which can be used to remove all entries with a particular value.

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

int main()
 {
  //Declare a list
  list<int> L;

  //Insert the number 100 at the beginning of the list
  L.push_front(100);

  //Insert the numbers 15 and 14 at the end of the list
  L.push_back(15); L.push_back(14);
  //Insert 14 at the front
  L.push_front(14);

  //Check the size of the list --- it should be 4
  cout << L.size();

  //Delete all entries with the value 14
  L.remove(14);
  //Now output the size --- it will be 2
  cout << " " << L.size() << endl;
 }
```

The above program produces the output `4 2`.

A list does not have the subscript operator `[]`, so iterators **must** be used to traverse the list.A subtlety is that the only operation that is permitted on a list iterator is the iteration operator `++`. The "random-access" operations `+=` and `-=` are not defined, so to find the iterator to the fourth entry in a list the following ugly loop must be used

```
list<int>::iterator p = L.begin();
for(int i=1;i<=4;i++) p++;
```

Lists are not designed for this! If you need to do this a lot, you are using the wrong data structure. A much more common procedure is to use the standard algorithm `find()` to find the iterator to the position of the first instance of a particular value in a sequence. It returns `end()` if there is no such value.

```
//Find the iterator to the value 100 in the list L
list<int>::iterator p = find(L.begin(), L.end(),100);
```

2.3 Map and Set

A map is a container whose values are referenced by a key, which is a generalised version of an array index. Each entry of the map has two values, the key (first) and the actual value (second). It is best illustrated by example

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

int main()
 {
  map<string, int> exam_result; //Entries are string, integer pairs

  //Set a couple of exam results
  exam_result["026578"] = 57;
  exam_result["5686"] = 10;

  //Traverse using iterators and print what we have
  for(map<string,int>::iterator p = exam_result.begin();
```

```
      p!=exam_result.end(); p++)
      {
        cout << "ID: " << p->first << " Result: " << p->second << endl;
      }
 }
```

The output is

```
ID: 026578 Result: 57
ID: 5686 Result: 10
```

A set is a collection of objects with the proviso that each value can occur only once. It can be regarded as a map with keys (which must also be unique), but no values. An example of the use of sets is

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

int main()
 {
  set<int> S; //An empty set of integers

  S.insert(10); //Add 10 to the set
  S.insert(5);  //Add 5 to the set
  S.insert(10); //10 is already in the set, so this will not be added

  //Number of elements in set is two
  cout << S.size() << endl;

  S.erase(5); //Delete the entry 5 from the set
 }
```

2.4 Stacks and Queues

A stack is a data structure that does three things efficiently: `top()` returns the entry at the top of the stack, `push()` adds an element to the top of the stack, and `pop()` removes

and item from the top of the stack.

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

int main()
 {
  stack<double> S; //An empty stack of doubles

  S.push(2.5);    //Add 2.5 to the stack
  S.push(3.1);    //Add 3.1 to the stack
  double x = S.top(); //Read the top entry from the stack, x = 3.1
  S.pop();         //Remove top entry, 3.1
 }
```

A queue is a data structure that supports efficient addition of an entry at the end of the queue `push()`, and removal at the front `pop()`.

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

int main()
 {
  queue<double> Q; //An empty queue of doubles

  Q.push(2.5);    //Add 2.5 to the end of the queue
  Q.push(3.1);    //Add 3.1 to the queue
  double x = Q.front(); //Read the first entry from the queue, x = 2.5
  double y = Q.back();  //Read the last entry from the queue y = 3.1
  Q.pop();         //Remove front entry, 2.5
 }
```

3 Standard Algorithms

A number of standard algorithms are provided in the C++ libraries, many of which are defined in the header `<algorithm>`. There are too many to cover here, but some of the most important are `sort()`, which sorts a sequence with good average efficiency, `reverse()`, which reverses the order of a sequence, and `rotate()`, which cyclically permutes a sequence. There are also the algorithms `max_element()`, which returns the iterator to the largest value in a sequence, `min_element()`, which returns the iterator to the smallest value in a sequence, and `count()`, which counts the number of occurrences of a value in a sequence. It is possible, in all cases, to overload the operator used for comparison, but the default is $>$.

An example code is shown below

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

int main()
 {
  //Define a vector and set the initial values
  vector<int> vec(5);
  vec[0] = -5; vec[1] = 3; vec[2] = -10; vec[3] = 1; vec[4] = 0;

  //Sort the vector in ascending numerical order
  sort(vec.begin(), vec.end()); // {-10,-5,0,1,3}
  //Now reverse the vector
  reverse(vec.begin(),vec.end()); //{3,1,0,-5,-10}
  //Now cyclically permute so that the vec.begin()+2 (third element)
  //becomes the first element in the array
  rotate(vec.begin(),vec.begin()+2,vec.end()); //{0, -5, -10, 3,1}

  for(int i=0;i<5;i++) cout << vec[i] << " ";
  cout << endl;

  cout << "Maximum element is " << *max_element(vec.begin(),vec.end()) << endl;
```



```
  cout << "Maximum element is " << *max_element(vec.begin(),vec.end()) << endl;
```


```
  cout << "Minimum element is " << *min_element(vec.begin(),vec.end()) << endl;
 }
```

There are plenty of other algorithms out there. If you are interested, you should look at some of the many books and articles on the C++ standard template library. Typing "C++ standard template library" into google also leads to lots of useful information.