Queens College, CUNY, Department of Computer Science

Object Oriented Programming in C++

CSCI 211 / 611

Summer 2018

Instructor: Dr. Sateesh Mane

July 22, 2018

STL: brief introduction

- This lecture contains a brief introduction to the **Standard Templates Library (STL).**
- We shall learn about a few types of **container classes.**
- We shall learn about **iterators.**
- We shall also learn about a few **algorithms.**

1 Introduction

- The **Standard Templates Library (STL)** is a very useful library in C++.
- As its name indicates, the STL is implemented entirely using templated classes.
- Both strings and vectors are part of the STL.
- Vectors are an example of a **container class.**
- As the name suggests, **container class** holds data values.
- We shall study **set** and **map**, which are also container classes.
- We shall learn about **iterators.**
- Iterators are employed to traverse a container class and access its elements.
- Iterators are a generalization of the concept of pointers.
- The STL also contains useful general purpose **algorithms.**
- We shall study only one algorithm, which is the sort function.

2 Set

- A **set** is a container class in the STL.
- To use a set, we include the header `#include <set>`.
- In a vector, the data elements are ordered in a linear sequence. We can access an element of a vector v via $v[i]$.
- By contrast, a set is a collection of data elements where the "index" is the value itself.
  1. We insert elements into a set, and they are **automatically sorted.**
  2. The data elements are not therefore pushed back onto the end of the set.
  3. The elements in a set are **unique.**
  4. If the same data value is inserted multiple times, only one copy is retained in the set.
- Here is a sample code to use a set, with data of type `string`.

```
set<string> s;          // set
s.insert("cx");         // "insert"
s.insert("dyz");
s.insert("aqt");        // elements are automatically sorted

s.insert("aqt");        // elements are unique, this replaces the previous entry
s.insert("bkmn");       // automatically sorted
```

- The elements of a set cannot be accessed using `operator[]`.
- The statement $s[i]$ will not compile.
- We access the elements of a set using an **iterator.**
- We shall study iterators below.

3 Map

- A **map** is a container class in the STL.
- To use a map, we include the header `#include <map>`.
- In a vector, we can access an element of a vector v via $v[i]$, where the index i is a nonnegative integer in the interval $0 \le i <$ `v.size`().
- By contrast, a map is a collection of (key, value) data elements, where the key is not necessarily an integer.
  1. We insert elements into a map by specifying a key and value.
  2. The data elements are **automatically sorted according to the key.**
  3. The data elements are not therefore pushed back onto the end of the map.
  4. The keys in a map are **unique.**
  5. If we insert a (key, value) pair (k, v_1) and then (k, v_2) with the same key, the first entry (k, v_1) will be erased from the map.
- Here is a sample code to use a map.
- The key is of type `string` and the value is of type `double`.

```
map<string,double> m;     // map, two template types for (key,value)
m["cde"]   = 1.2;         // key, value
m["xyz"]   = 3.4;         // key, value
m["abcd"]  = 5.6;         // key, value, automatically sorted by key

m["abcd"]  = 7.8;         // key, value, replaces previous entry
m["bmnsy"] = 9.9;         // key, value, automatically sorted by key
```

- The elements of a map can be accessed using `operator[]`, but the index is the key, which need not be an integer.
- We access the elements of a map using an **iterator.**
- We shall study iterators below.

4 Vectors, sets, maps

- Vectors, sets and maps are all example of container classes.
- They all contain data, but in different ways.
- In terms of a (key, value) relationship, elements in a vector or a map are indexed by a key, while for a set the key and the value are the same,
- For a vector v, the key of an element $v[i]$ is an integer i.
  1. The value of the key $(= i)$ is an integer, and it spans the interval $0 \leq i <$ `v.size`().
  2. We cannot create a vector of two elements with entries $(v[10], v[1000])$ and nothing else.
  3. For a vector with two elements, the entries are necessarily $(v[0], v[1])$.
  4. The data type of the value a vector can be anything, including a user-defined class.
- For a map m, the key of an element need not be an integer.
  1. We can create a map of two elements with entries $(m[10], m[1000])$ and nothing else.
  2. But we can also create a map of two elements with entries `(m["abc"],m["xyzt"])`.
  3. We can also create a map of two elements with entries $(m[-1.234], m[77.345])$.
  4. *Both data types for the key and value of a map can be anything*, including user-defined classes.
- For a set s, the key and value of an element are the same thing.
  1. Since the key of an element must be unique, the values in a set are therefore unique.
  2. For a set, the key/value of an element need not be an integer.
  3. The data type of the key/value of a set can be anything, including a user-defined class.
- See Sec. 9 for comments about using user-defined classes for sets and maps.

5 Iterators

- For a vector v, we can traverse all the elements by writing a loop as follows.

```
for (int i = 0; i < v.size(); ++i)
  cout << v[i] << endl;
```

- However, the above loop does not work for sets and maps.
- There is way to traverse all container classes in the STL, which is to employ **iterators.**
  1. An iterator is a generalization of a pointer.
  2. We shall only study a few details about iterators.
  3. Essentially only to access elements in an STL container class.
- The following code shows how to use an iterator to traverse a vector and print the values of the elements. The result is the same as the previous loop.

```
vector<string> v;                                   // vector of data type string
// populate the vector
vector<string>::iterator v_it;                      // iterator
for (v_it = v.begin(); v_it != v.end(); ++v_it)     // end of loop test, increment iterator
    cout << *v_it << endl;                          // dereference of iterator
```

- Note the following:
  1. Just as a pointer is specific to a data type (pointer to `int`, pointer to `double`, etc.), an iterator is also specific to a data type.
  2. For `vector<string>`, the iterator is `vector<string>::iterator v_it;`
  3. For `vector<int>`, the iterator would be `vector<int>::iterator v_it;`
  4. We initialize the iterator to point to the start of the vector `v_it = v.begin()`.
  5. Here `v.begin()` is itself an iterator, which points to the start of the vector.
  6. The end-of-loop condition is `v_it != v.end()`.
  7. Here `v.end()` is an iterator which points to **one place beyond the end of the vector.**
  8. This is analogous to the end-of-loop condition `i < v.size()`, because the key goes up to `v.size()-1`.
  9. The iterator increment condition `++v_it` (we can also write postfix `v_it++`).
  10. To access a data element in the vector, we dereference the iterator `*v_it`.
  11. The notation is similar to the dereference of a pointer.
  12. **We can assign the data elements in the vector by writing**

```
*v_it = (value);
```

5.1 Set iterator

- For a set s, the analogous loop is as follows.

```
set<string> s;
// populate the set
set<string>::iterator s_it;                    // iterator for set
for (s_it = s.begin(); s_it != s.end(); ++s_it)   // end of loop test, increment iterator
    cout << *s_it << endl;                     // dereference of iterator
```

- The declaration of the iterator for a set is similar to that for a vector.
  1. One technicality is that we cannot assign `*s_it = (value)`.
  2. This is because the key and value of a set are the same.
  3. Hence changing the value would change the key and the object pointed to by the iterator would no longer exist.
  4. This would **invalidate the iterator.**

5.2 Map iterator

- For a map m, the analogous loop is as follows.

```
map<string, double> m;
// populate the map
map<string, double>::iterator m_it;                 // iterator for map
for (m_it = m.begin(); m_it != m.end(); ++m_it) {   // end of loop test, increment
    m_it->second = m_it->second + 2.468;            // assignment of value
    cout << m_it->first << " " << m_it->second << endl;  // arrow operator
}
```

- The declaration of the iterator for a map is similar to that for a vector.
  1. However, a map has a key and a value.
  2. The map iterator contains two public data members `first` and `second`.
  3. Obviously "first" is the key and "second" is the value.
  4. We can assign the value `m->second = (something)`.
  5. Obviously we cannot assign the key. To do so would invalidate the iterator.

6 Traversing backwards

- For a vector v, we can traverse all the elements in reverse order by writing a loop as follows.

```
for (int i = v.size()-1; i >= 0; --i)
  cout << v[i] << endl;
```

- There are complications, to traverse the elements in reverse order using the above iterators.
- Here is the code to traverse a vector in reverse order.

```
vector<string>::iterator v_it;
for (v_it = v.end(); v_it != v.begin(); --v_it) {   // decrement the iterator
  vector<string>::iterator tmp = v_it;
  if (tmp != v.begin()) --tmp;
  cout << *tmp << endl;
}
```

- The code is messy.
  1. Similar to the integer, we decrement the iterator `--v_it`.
  2. However, the traversal begins at `end()` and terminates when the iterator equals `begin()`.
  3. ***But this is not what we want.***
  4. First, `end()` points to one place beyond the end of the vector. We do not want to print an invalid value.
  5. Next, the loop terminates when `v_it == begin()`. But we want to print that value.
  6. Hence we define a temporary iterator variable `tmp`.
  7. We test if `tmp` equals `v.begin()` and if not, we decrement `tmp` one unit and print the dereference `*tmp`.
- **This solution works, but it is terrible.**
- There is a better way, and that is to employ a **reverse iterator**.

7 Reverse iterators

- **The previous iterators are all forward iterators.**
- To traverse the elements of a container class in reverse order, STL provides **reverse iterators.**
- Here is the code to use reverse iterators for a vector, set and map.

```
vector<string> v;
// populate the vector
vector<string>::reverse_iterator v_rit;                // reverse iterator
for (v_rit = v.rbegin(); v_rit != v.rend(); ++v_rit)   // end of loop test, increment
  cout << *v_rit << endl;                              // dereference

set<string> s;
// populate the set
set<string>::reverse_iterator s_rit;                   // reverse iterator
for (s_rit = s.rbegin(); s_rit != s.rend(); ++s_rit)   // end of loop test, increment
  cout << *s_rit << endl;                              // dereference

map<string, double> m;
// populate the map
map<string, double>::reverse_iterator m_rit;             // iterator
for (m_rit = m.rbegin(); m_rit != m.rend(); ++m_rit) {   // end of loop test, increment
  m_rit->second = m_rit->second + 3.456;                 // assignment
  cout << m_rit->first << "  " << m_rit->second << endl; // arrow operator
}
```

- Note the following:
  1. Reverse iterators traverse from `rbegin()` ("reverse begin") to `rend()` ("reverse end").
  2. **Note that `rbegin()` is not the same as `end()-1`, and `rend()` is not the same as `begin()-1`.**
  3. Forward and reverse iterators are different classes of iterators.
  4. Just as we cannot mix and match pointers of different types, we cannot mix and match forward and reverse iterators.
  5. **Note that we increment the iterators using "++" (not "−−").**
  6. The "reverse" definition deals with the direction of traversal. Don't be too clever.

8 `const` iterators

- *Yes there are* `const` *iterators!*
- *And there are* `const` *reverse iterators!*
- A `const` forward or reverse iterator does not change the values of the data elements.
- In our example, where we only print the values of the data elements, `const` forward/reverse iterators can be employed.
- A `const` forward iterator traverses from **cbegin()** to **cend()**.
- A `const` reverse iterator traverses from **crbegin()** to **crend()**.
- `const` forward iterators:

```
vector<string>::const_iterator v_cit;
for (v_cit = v.cbegin(); v_cit != v.cend(); ++v_cit)
  cout << *v_cit << endl;

set<string>::const_iterator s_cit;
for (s_cit = s.cbegin(); s_cit != s.cend(); ++s_cit)
  cout << *s_cit << endl;

map<string,double>::const_iterator m_cit;
for (m_cit = m.cbegin(); m_cit != m.cend(); ++m_cit)
  cout << m_cit->first << "  " << m_cit->second << endl;
```

- `const` reverse iterators:

```
vector<string>::const_reverse_iterator v_crit;
for (v_crit = v.crbegin(); v_crit != v.crend(); ++v_crit)
  cout << *v_crit << endl;

set<string>::const_reverse_iterator s_crit;
for (s_crit = s.crbegin(); s_crit != s.crend(); ++s_crit)
  cout << *s_crit << endl;

map<string, double>::const_reverse_iterator m_crit;
for (m_crit = m.crbegin(); m_crit != m.crend(); ++m_crit)
  cout << m_crit->first << "  " << m_crit->second << endl;
```

9 User-defined class

- Let us create a user-defined class and create sets and maps for it.
- As a simple example, consider the folliwing class `MyString`.

```
class MyString {
public:
  MyString() {}
  MyString(const string &a) { str = a; }
  string str;
};
```

- It is just a wrapper around a `string`.
- However, the class `MyString` is lacking a significant feature.
- **The comparison "`operator<`" does not exist for the class `MyString`.**
- Therefore `MyString` objects **cannot be sorted.**
- *Therefore it is impossible to create sets of* `MyString` *objects.* A set is a sorted collection, and the compiler does not know how to sort a collection of `MyString` objects.
- We can create `vector<MyString>`, because objects in a vector are not automatically sorted.
- We can create a map, if the class `MyString` is the value.
- However, we cannot create a map if the class `MyString` is the key.
- The keys of a map are automatically sorted, and the compiler does not know how to sort a collection of `MyString` objects.
- **We can create a set for a user-defined class only if the comparison `operator<` is overloaded for that class.**
- **We can create a map and use a user-defined class as the key only if the comparison `operator<` is overloaded for that class.**
- Here is the overloaded `operator<` for the class `MyString`.

```
bool operator< (const MyString &u, const MyString &v)
{
  return (u.str < v.str);
}
```

- If `operator<` is overloaded for the class `MyString`, then we can do the following:
  1. We can create `set<MyString>`.
  2. We can use `MyString` as the key for `map<MyString, (value)>`.

10 Algorithms

- The STL also provides a collection of general purpose **algorithms.**
- To use the STL algorithms we include the header `#include <algorithm>`.
- We shall consider only one algorithm, which is `sort`.
- The function `sort` takes three input parameters.
  1. An iterator to the start of the collection to be sorted.
  2. An iterator to the one place beyond the end of the collection to be sorted.
  3. A third parameter for a comparison function.
- The third parameter is optional for data types such as `int` or `double` (or `string`), which the compiler already knows how to sort.
- Here is an example main program to employ `sort` for a vector of `MyString` objects.

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

class MyString {                                       // user defined class
public:
  MyString() {}
  MyString(const string &a) { str = a; }
  string str;
};

bool operator< (const MyString &u, const MyString &v)  // comparison operator
{ return (u.str < v.str); }

int main()
{
  vector<MyString> v;
  v.push_back(MyString("Charlie"));
  v.push_back(MyString("Alice"));
  v.push_back(MyString("Bob"));
  sort(v.begin(), v.end(), operator<);                 // STL sort function

  for (int i=0; i < v.size(); ++i)
    cout << v[i] << endl;

  return 0;
}
```