CS 101:
Computer Programming and Utilization

Jul-Nov 2017

Umesh Bellur
(cs101@cse.iitb.ac.in)

Lecture 15: Standard Library

About These Slides

- Based on Chapter 22 of the book

An Introduction to Programming Through C++

by Abhiram Ranade (Tata McGraw Hill, 2014)

- Original slides by Abhiram Ranade
  - First update by Uday Khedker
  - Second update by Sunita Sarawagi

The Standard Library

- Comes with every C++ distribution
- Contains many functions and classes that you are likely to need in day to day programming
- The classes have been optimized and debugged thoroughly
- If you use them, you may be able to write programs with very little work
- Highly recommended that you use functions and classes form the standard library whenever possible

Outline

- The string class
- The template class vector
  - Multidimensional vectors
  - Sorting a vector
- The template class map
  - Iterators
- Remarks

The String Class

- A much more powerful version of the String class developed in Chapter 21
- More constructors
- Concatenation using +
- Works with >> and <<
- Operations for extracting substrings and finding one string inside another

Examples

```
#include <string>      // Needed to use the string class
string v = “abcdab”;   // constructor
string w(v);           // another constructor. w = v
v[2] = v[3];   // indexing allowed. v becomes “abddab”
cout << v.substr(2)  << v.substr(1,3)  << endl;
        // substring starting at v[2] (“ddab”)
        // Substring starting at v[1] of length 3 (“bdd”)
int i = v.find(“ab”);      // find occurrence of “ab” in v
                           // and return index
int j = v.find(“ab”,1);   // find from index 1
cout << i << “, “ << j << endl; // will print out 0, 4.
```

Remarks

- If the find member function does not find the argument in the receiver, then it returns a constant string::npos, which is a value which cannot be a valid index
  - You can determine whether the argument was found by checking whether the returned index equals string::npos
- string s; s.size() or s.length() returns a value of type size_t (a redefinition of unsigned int)
- A string object can be passed by value, in which case it is copied, or by reference
- More details on the web. Example: http://www.cplusplus.com/reference/string/

Templates

- Function templates (Sec 12.5 in book)
- Consider these three functions: same body, different types

`int Abs(int x) {`
`  if (x < 0)`
`    return -x;`
`  else return x;`
`}`
	`float Abs(float x) {`
	`  if (x < 0)`
	`    return -x;`
	`  else return x;`
	`}`
		`double Abs(double x) {`
		`  if (x < 0)`
		`    return -x;`
		`  else return x;`
		`}`

A common template to unite them all....

```
template<typename T>
T Abs(T x) {
  if (x < 0)
    return -x;
  else return x;
}
```

Template Class

- Like function templates, create class with templates.

template <class T>
class Queue {
  int front, nWaiting;
  T elements[100];
public:
  bool insert(T value)
{…}
  bool remove(T &val)
{…}
};
	main () {
	  Queue<V3> q;
	  Queue<int> r;
	}

The Template Class Vector

- Friendlier, more versatile version of arrays
- Must include header file <vector> to use it
- You can make vectors of any type by supplying the type as an argument to the template
- Indexing possible like arrays
- Possible to extend length, or even insert in the middle
- We will not discuss how the vector class is implemented, but you should be able to guess that its member functions would allocate memory and deallocate it as needed

Examples

```
#include <vector>// needed
vector<int> v1;     //empty vector.  Elements will be int
vector<float> v2;  //empty vector. Elements will be float
vector<short> v3(10); // vector of length 10.
                        //   Elements are of type short
vector<char> v4(5,'a');     // 5 elements, all 'a'
cout << v3.size() << endl; // prints vector length, 10
                            // v3.length() is same
v3[6] = 34;                 // standard indexing
```

Examples (Contd.)

```
#include <vector>            // needed
…

v3.push_back(22);            // append 22 to v3.
                             // Length increases
vector<char> w;
w = v5;                      // element by element copy
v1.resize(9);                // change length to 9
v2.resize(5, 3.3);           // length becomes 5, all
                             // values become 3.3
vector<string> s;            // vector of string
vector<vector<int> > vv;     // allowed!
```

A Technical Remark

- The member function size returns a value of type size_t
- size_t is an unsigned integer type; it is meant specially for storing array indices
- When going through array elements, use size_t for the index variable

```
vector<double> v(10);          // initialize v
for(size_t i=0; i<v.size(); i++)
    cout << v[i] << endl;
```

- If i were declared int, then the compiler would warn about the comparison between i and v.size()
  - comparison between signed and unsigned int, which is tricky as discussed in Section 6.8.
  - By declaring i to be size_t, the warning is suppressed.

Multidimensional Vectors

```
vector<vector <int> > vv;
// each element of vv is itself a vector of int
// we must supply two indices to get to int
// Hence it is a 2d vector!
// Currently vv is empty
vector<vector <int> > vv1(5, vector<int>(10,23));
// vv1 has 5 elements
// each of which is a vector<int>
// of length 10,
// having initial value 23
```

Multidimensional Vectors

- Note that the syntax is not new/special
- It is merely repeated use of specifying the length and initial value:
- vector<type> name(length, value)
- Two dimensional arrays can be accessed by supplying two indices, i.e. we may write vv1[4][6] and so on
- Write vv1.size() and vv1[0].size() to get number of rows and columns

Creating A 5x5 Identity Matrix

```
vector<vector<double>> m(5, vector<double>(5,0));
                        // m = 5x5 matrix of 0s
                        // elements of m can be accessed
                        // by specifying two indices
for(int i=0; i<5; i++)
  m[i][i] = 1;
                        // place 1s along the diagonal
```

Remarks

- The book gives a matrix class which internally uses vector of vectors
- This class is better than two dimensional arrays because it can be passed to functions by value or by reference, with the matrix size being arbitrary

Sorting A Vector

- C++ provides a built-in facility to sort vectors and also arrays
- You must include <algorithm> to use this

```
vector<int> v(10);
// somehow initialize v
sort(v.begin(), v.end());
```

- That's it! v is sorted in non decreasing order
- begin and end are "iterators" over v. Think of them as abstract pointers to the beginning and the end.

Sorting An Array

- The algorithms in header file <algorithm> can also sort arrays as follows

```
double a[100];
// somehow initialize a
sort(a, a+100);  // sorted!
// second argument is name+length
```

- More variations in the book

The Map Template Class

- A vector or an array give us an element when we supply an index
  - Index must be an integer
- But sometimes we may want to use indices which are not integers, but strings
  - Given the name of a country, we may want to find out its population, or its capital
  - This can be done using a map

Map: General Form And Examples

- General form:
  map<indextype, valuetype> mapname;

- Examples:
  map<string,double> population;
  Indices will have type string (country names), and elements will have type double (population)

  map<string, vector<string>> dictionary;
  Maps words to their meanings expressed as a vector of other words.

Using A Map

```
map<string,double> population;

population["India"] = 1.21;
                    // in billions.  Map entry created
population["China"] = 1.35;
population["USA"] = 0.31;

cout << population["China"] << endl;
                    // will print 1.35

population["India"] = 1.22;
                    //update allowed
```

Checking if An Index is Defined

```
string country;
cout << “Give country name: “;
cin >> country;

if(population.count(country)>0)
            // true if element with index = country
            // was stored earlier
            // count is a known member function
  cout << population[country] << endl;
else cout << “Not known.\n”;
```

Remarks

- A lot goes on behind the scenes to implement a map
- Basic idea is discussed in Chapter 24 of our book
- If you wish to print all entries stored in a map, you will need to use iterators, discussed next

Iterators

- A map can be thought of as holding a sequence of pairs, of the form (index, value)
- For example, the population map can be considered to be the sequence of pairs

  [(“China”,1.35), (“India”,1.21), (“USA”, 0.31)]
- You may wish to access all elements in the map, one after another, and do something with them
- For this, you can obtain an iterator, which points to (in an abstract sense) elements of the sequence

Iterators

An iterator points to (in an abstract sense) elements of the sequence

- An iterator can be initialized to point to the first element of the sequence
- In general, given an iterator which points to some element, you can ask if there is any element following the element, and if so make the iterator point to the next element
- An iterator for a map<index,value> is an object with type map<index,value>::iterator

Iterators (contd.)

- An iterator points to elements in the map; each element is a struct with members first and second
- We can get to the members by using dereferencing
- Note that this simply means that the dereferencing operators are defined for iterators
- If many elements are stored in an iterator, they are arranged in (lexicographically) increasing order of the key

Example

```
map<string,double> population;
population["India"] = 1.21;

map<string,double>::iterator mi;
mi = population.begin();
            // population.begin() : constant iterator
            // points to the first element of population
            // mi points to (India,1.21)
cout << mi->first << endl; // or (*mi).first << endl;
            // will print out India
cout << mi->second << endl;
            // will print out 1.21
```

Example

```
map<string,double> population;
population["India"] = 1.21;
population["China"] = 1.35;
population["USA"] = 0.31;
for(map<string,double>::iterator
    mi = population.begin();
    mi != population.end();
                    // population.end() : constant iterator
                    // marking the end of population
     mi++)
                    // ++ just sets mi to point to the
                    // next element of the map
     // loop body
```

Example (Contd.)

```
map<string,double> population;
population[“India”] = 1.21;
population[“China”] = 1.35;
population[“USA”] = 0.31;
for(map<string,double>::iterator
    mi = population.begin();
    mi != population.end();
    mi++)
{
  cout << (*mi).first << “: “ << (*mi).second << endl;
  // or cout << mi->first << “: “ << mi->second << endl;
}
// will print out countries and population in alphabetical order
```

Remarks

- Iterators can work with vectors and arrays too
- Iterators can be used to find and delete elements from maps and vectors.

```
map<string,double>::iterator
      mi = population.find("India");
population.erase(mi);
```

Map with user-defined class as index

- Any class used as indextype on a map must implement the "<" operator.
- Example, the following code will not work because "<" is not defined on V3.
  - class V3 {public: double x,y,z};
  - map<V3, string> vec2string;
- A correct implementation of V3 may be something like:

```
class V3 {
    public:
    double x,y,z;
    bool operator<(const V3& a) const {
        if (x < a.x) return true;
        if (x == a.x && y < a.y) return true;
        if (x==a.x && y == a.y && z < a.z) return true;
        return false;
    }
};
```

Sets

- Sets are containers that store unique elements following a specific order
- The value of the elements in a set cannot be modified once in the container (the elements are always const), but they can be inserted or removed from the container
- Internally, the elements in a set are always sorted following a specific ordering criterion indicated by its internal comparison object

Populating and Traversing a Set

```
#include <set>                    // set class library
 …
 set<int> set1;              // create a set object,
                                   // specifying its content as int
                                  // the set is empty
int ar[]={3,2,4,2};
for (int i = 0; i < 4; i++) {
  set1.insert(ar[i]);    // add elements to the set.
}
for (set<int>::iterator iter = set1.begin(); iter != set1.end(); iter+
+) {   cout << *iter << "  ";
}  // prints 2 3 4
```

Application of Set

Given N students where each student has a list of courses that they have taken.
Create group of all students that have taken exactly the same set of courses.

```
map<set<string>, vector<int>> study_group;
  // key of the map is the set of courses.
  // value is vector of student roll-numbers of students who
have taken this course.
   cin >> N;
   for(int i = 0; i < N; i++) {
      int roll, int n;
      cin >> roll >> n;
       set<string> subjects;
```

Application of Set (continued)

```
    for (int j = 0; j < n; j++) {
        string s; cin >> s;
        subjects.insert(s);
    }
    study_group[subjects].push_back(rollno);
}
```

List

- Implements a classic list data structure
- Supports a dynamic bidirectional linear list
- Unlike a C++ array, the objects the STL list contains cannot be accessed directly (i.e., by subscript)
- Is defined as a template class, meaning that it can be customized to hold objects of any type
- Responds like an unsorted list (ie. the order of the list is not maintained). However, there are functions available for sorting the list

Populating and Traversing a List

```
#include <list>                     // list class library
 …
 list <int> list1;            // create a list object,
                                    // specifying its content as int
                                   // the list is empty
for (i=0; i<5; i++)
  list1.push_back (i);    // add at the end of the list
…
while (list1.size() > 0)
{   cout << list1.front();       // print the front item
     list1.pop_front();            // discard the front item
}
```

Concluding Remarks

- Standard Library contains other useful classes, e.g. queue, list, set etc.
- The Standard Library classes use heap memory, however this happens behind the scenes and you don't have to know about it
- The library classes are very useful. Get some practice with them
- More details on the web. Example: http://www.cplusplus.com/reference/stl/