# LECTURE 15

## The Standard Library (STL)

We have been dealing with input and output streams which are classes in the standard library. The standard library has many useful classes and features besides input and output. These include classes for strings, lists, vectors, maps, complex numbers, etc. Details can be found in Stroustrup's book (Chapters 3, and 16-22). Before we talk about the standard library, let me explain what a namespace is.

### Namespaces

We have discussed the benefits of modular programming in which one divides up the program into modules. Namespaces assist in doing this. Related data, functions, classes, etc. can be grouped into separate namespaces. For example, a namespace declaration would look like:

```
namespace Matrices{
   class 1DArray;                //classes
   class 2DArray;
   1DArray operator +(const 1DArray& a, const 1DArray& b);  //functions
   2DArray operator +(const 2DArray& a, const 2DArray& b);  //functions
   ostream& display(const 1DArray& a);                      //functions
   ostream& display(const 2DArray& a);                      //functions

}
```

To refer to a member of this namespace, one uses the scope resolution operator, e.g., `Matrices::display(a)` and `Matrices::1DArray`.

The standard library is defined in a namespace called `std`. So if you refer to a type (class) or function from the standard library, like `string`, you should write `std::string`.

### Including the Standard Library in Your Code

Every standard library facility is provided through some standard header similar to `<iostream>` (notice the absence of "`.h`"). For example,

```
#include<string>
#include<list>

std::string s = "Four legs Good; two legs Baaaad!";
std::list<std::string> slogans;
```

Many of the classes in the standard library are template classes. For example, `list` is a template class. The standard library is sometimes called the standard template library (STL).

Writing `std::` as a prefix is a pain; so one can dump the names into the global namespace in the following way:

```
#include<string>          //make the standard string facilities accessible
```

```
using namespace std;      //make std names available without std:: prefix

string s = "Ignorance is bliss!"   //ok; sting is std::string
```

It is generally in poor taste to dump every name from a namespace into the global namespace, because you run the risk of name clashes if you name a variable the same as something in the standard library. You can dump the names from a namespace into a local space, e.g., you could make the names local to a function:

```
#include<string>          // complex number facility

std::string function()
{
  using namespace std;      //make std names available without std:: prefix
  string s = "Ignorance is bliss!";   //ok; sting is std::string
  //other stuff
}
```

(Actually many compilers like g++ just assume the standard library names are in the global namespace, so you don't have to write "`using namespace std`." But this may change as compilers are upgraded.)

As we mentioned earlier, the iostream classes are in the standard library. So we could write,

```
#include<iostream>

 std::cout << "hello! \n";
```

We have just been writing `cout` because we included `iostream.h`. In some cases, such as `iostream`, by including `<X.h>` rather than `<X>`, we make the `std` names global. `iostream` and `fstream` are examples of this. So are headers in the C standard library. A standard header with a name starting with the letter `c` is equivalent to a header in the C standard library. For every header `<cX>` defining names in the `std` namespace, there is a header `<X.h>` defining the same names in the global namespace. For example, the standard math library facility `<cmath>` corresponds to the C library `<math.h>`. We can write

```
#include<cmath>

 void main(){
   float x=2;
   float a;
   a = std::pow(x,4);
 }
```

or we can dump the names in the global namespace:

```
#include<math.h>

 void main(){
   float x=2;
   float a;
   a = pow(x,4);          //ok; refers to std::pow
 }
```

By the way, blank spaces matter in include statements. For example, the following will not work:

```
#include< iostream.h >
```

The standard library has facilities that deal with

**input–output** (e.g., `<iostream>`)

**strings** (e.g., `<string>`)

**containers** (e.g., `<vector>`, `<list>`, `<map>`, etc.)

**iterators** move and search through the containers

**algorithms** do things like copy, sort, find the occurrence of an argument, loop through a collection of elements and perform a function on each one

**numerics** (e.g., complex numbers, numeric vectors and operations on them, random numbers, standard mathematical functions)

A list of the facilities of the standard library can be found in Chapter 16 of Stroustrup. There are entire books written on the standard library. We don't have the time to examine the standard library in detail. (There are entire books written on the standard library.) But I want you to be aware of its existence. Let us give some brief examples of these facilities. (We will drop the `std::` prefix for convenience.)

**Strings**

The standard library provides a `string` type to complement the string literals (`char*`) used earlier. The `string` class provides a variety of useful string operations, such as concatenation of strings with the `+` sign. For example:

```
#include <string>

string s1 = "Hello";
string s2 = "world";

void func()
  {
```

```
    string s3 = s1 + "," + s2 + "!\n";
    cout << s3;
  }
```

s3 is initialized to `Hello, world!` followed by a newline. For strings, addition means concatenation. You can add strings, string literals, and characters to a string.

In many applications, the most common form of concatenation is adding something to the end of a string. This is directly supported by the `+=` operation. For example:

```
#include <string>

void func(string& s1, string& s2)
  {
    s1 = s1 + '\n';      //append newline
    s2 += '\n';          //append newline
  }
```

These two ways of adding to the end of a string are semantically equivalent.

Naturally, `string`s can be compared against each other and against string literals. For example:

```
#include <string>

string password;

void respond(const string& answer)
 {
   if (answer==password)
     { // do something}

   else if (answer=="yes")
     { // do something else}
 }
```

The string class also provides the ability to manipulate substrings. For example,

```
#include <string>

string name = "George Washington";

void func()
  {
    string s = name.substr(7,10);      //s = "Washington"
    name.replace(0,6,"Mrs. Martha");  //name becomes "Mrs. Martha Washington"
  }
```

The `substr()` operation returns a string that is a copy of the substring indicated by its arguments. The first argument is the position of the first letter of the substring, and the second argument is the length of the substring. Since indexing starts from 0, `s` gets the value of "Washington".

The replace operation replaces a substring with a value. In this case the substring starting at 0 with length 6 is `George`; it is replaced by `Mrs. Martha`. Thus the final value of `name` is `Mrs. Martha Washington`. Note that the replacement string need not be the same size as the substring that it is replacing.

**Math**

The standard library has a mathematical component to it that supports complex numbers, vector arithmetic, mathematical functions, etc. Details can be found in Stroustrup, Chapter 22.

**Complex Numbers**

A complex number has the form $a + ib$ where $a$ and $b$ are real numbers. They could be `floats` or `doubles`, etc. So the complex numbers are a templated class that allows you to choose.

```
#include<complex>

template<class scalar>
class complex
{
  public:
   complex(scalar re, scalar im);
   // ...
};
```

The usual arithmetic operations and the most common mathematical functions are supported for complex numbers. For example:

```
#include<complex>

template<class C>
complex<C> pow(const complex<C>&, int);    //raise complex number to a power

template<class C>
complex<C> cosh(const complex<C>&);        //cosh

void f(complex<float> fl, complex<double> db)
  {
    complex<long double> ld = fl + sqrt(db);
    db += fl*3;
    fl = pow(1/fl,2);
```

```
    // ...
   }
```

More details can be found in Stroustrup, section 22.5.

**Vector Arithmetic**

Vectors are used all the time in physics. `<valarray>` supports vector arithmetic. For example,

```
#include<valarray>

template<class T>
class valarray
 {  // ...
   T& operator[](size_t);
    // ...
 }
```

The type `size_t` is the unsigned integer type that the implementation uses for array indices.

The usual arithmetic operations and the most common mathematical functions are supported for `<valarray>`s. For example:

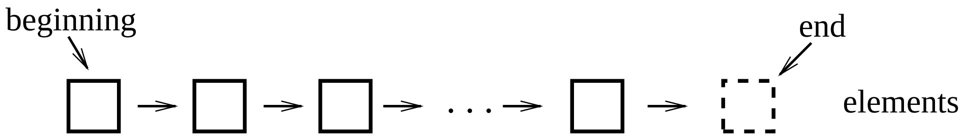
```
#include<valarray>

template<class T>
valarray<T> abs(const valarray<T>&);       // absolute value

void func(valarray<double>& a1, valarray<double>& a2)
{
  valarray<double> a = a1 * 3.14 + a2/a1;
  a2 += a1 * 3.14;
  a = abs(a);
  double d = a2[7];
  // ...
}
```

More details can be found in Stroustrup, section 22.4.

**Containers**

Much computing involves creating collections of objects and then manipulating such collections. A class whose main purpose is holding objects is called a *container* or a container class. An example of a container is the linked list that we studied earlier in a homework problem. The standard library has container classes such as `vector`, `list`, `map`, etc.

Container share the notion of a sequence. We can represent a sequence graphically like this:




A sequence has a beginning and an end. An iterator is sort of like a pointer to an element. An iterator refers to an element and provides an operation that makes the iterator refer to the next element of the sequence. The end of a sequence is an iterator that refers one beyond the last element of the sequence. The physical representation of "the end" may be a sentinel element (e.g. `'\0'` which ends `char*` strings), but it doesn't have to be. We need some standard notation for operations such as "access an element through an iterator" and "make the iterator refer to the next element." The obvious choices (once you get the idea) are to use the dereference operator `*` to mean "access an element through an iterator" and the increment operator `++` to mean "make the iterator refer to the next element."

Let's examine some of the container classes:

**Vector**

A built–in array has a fixed size and it's not easy to change the size. For example, recall the class `Point` that we dealt with in lecture 3. If we make an array of `Point`s by writing

```
class Point;
Point pts[100];
```

then it's hard to resize the array `pts` later in the program. The standard library provides a templated `vector` class to take care of that:

```
#include <vector>

class Point;
vector<Point> pts(100);      //Note use of parentheses
void display(int i)          //simple use; exactly as for an array
  { cout << pts[i].x() << endl; }  // Print x-coordinate

void add_entries(int n)     //increase size by n
  { pts.resize(pts.size() + n); }
```

The vector member function `size()` gives the number of elements. Note the use of parentheses in the definition of `pts`. We made a single object of type `vector<Point>` and supplied its initial size as an initializer. This is very different from declaring a built-in array:

```
vector<Point> point(1000);   //vector with 1000 points
vector<Point> points[1000];  //1000 empty vectors
```

Should you make the mistake of using `[ ]` where you meant `()` when declaring a `vector`, your compiler will almost certainly catch the mistake and issue an error message when you try to use the `vector`.

A `vector` is a single object that can be assigned. In other words, the assignment operator for `vector` has been overloaded. For example:

```
#include <vector>

class Point;
vector<Point> pts(100);

void func(vector<Point>& v)
 {
   vector<Point> v2 = pts;
   v = v2;
   // ...
 }
```

Assigning a `vector` involves copying its elements. Thus, after initialization and assignment in `func`, `v` and `v2` each holds a separate copy of every `Point` in `pts`. When a `vector` holds many elements, such innocent–looking assignments and initializations can be prohibitively expensive. Where copying is undesirable, references or pointers should be used. Or one can use reference counting as we described in lecture 13.

**Example of Using Standard Classes as Base Classes: Bounds Checking**

We can use the standard library classes as base classes, i.e., we can write our own classes that are derived from the standard library classes. As an example of this, we can derive a class from `vector` that provides bounds checking. The standard library `vector` does not provide bounds checking by default. Suppose we write

```
#include <vector>

class Point;
vector<Point> pts(100);
void func()
  { double xx = pts[101].x(); }  // 101 is out of bounds
```

The initialization of `xx` will likely give some garbage value to `xx` rather than giving an error. This is undesirable, so we can make a derived class `Vec` that will do bounds checking. A `Vec` is like a `vector`, except that it throws an exception of type `out_of_range` if a subscript is out of bounds.

```
#include <vector>

template<class T>
```

```
class Vec: public vector<T>    //Vec is derived from vector
{
  public:
    Vec() : vector<T>() {}
    Vec(int s) : vector<T>(s) {}

    T& operator[](int i) {return at(i);}               //bounds checked
    const T& operator[](int i) const {return at(i);}   //bounds checked
};
```

The at() operation is a vector subscript operation that throws an exception of type out_of_range if its argument is out of the vector's range. Notice that Vec is derived from vector. We use Vec in exactly the same way as we used vector.

```
#include <vector>
class Point;
Vec<Point> pts(100);
void display(int i)           //simple use; exactly as for a vector
  { cout << pts[i].x() << endl; }  // Print x-coordinate
```

An out of bounds request will throw an exception that the user can catch. For example:

```
void f()
{
  try
   {
     for(int i=0; i<1000; i++)
       display(i);
   }
  catch(out_of_range)
   {
     cout << "range error\n";
   }
}
```

**List**

A list is another container of objects. When creating a list object, it has the similar syntax to vector's:

```
#include <list>

class Point;
list<Point> pts;
```

It differs from a vector in a few respects. For example, it is easier to add and delete entries from a list than from a vector. When we use a list, we tend not to access elements using subscripting the way we commonly do for vectors. Instead, we might search a list looking for an element with a given value. To do this, we take advantage of the fact that a list is a sequence as we described earlier. Recall that a sequence has an iterator to refer to an element and provides an operation that makes the iterator refer to the next element of the sequence. The dereference operator * is used to "access an element through an iterator" and the increment operator ++ is used to "make the iterator refer to the next element."

```
#include <list>

class Point;
list<Point> pts;

void print_point(const float& xx, const float& yy)
  {
    typedef list<Point>::const_iterator LI;
    for (LI i=pts.begin(); i != pts.end(); ++i)
     {
       Point& p = *i;              //reference used as shorthand
       if((xx == p.x()) && (yy == p.y()))
         cout << p.x() << ' ' << p.y() << '\n';
     }
  }
```

The search for the point with the coordinates (xx,yy) starts at the beginning of the list and proceeds until either (xx,yy) are found or the end is reached. Every standard library container provides the functions begin() and end(), which return an iterator to the first and to one–past–the–last element, respectively. Given an iterator i, the next element is ++i. Given an iterator i, the element it refers to is *i.

A user need not know the exact type of the iterator for a standard container. That iterator type is part of the definition of the container and can be referred to by name. When we don't need to modify an element of the container, const_iterator is the type we want. Otherwise, we use the plain iterator type.

Adding elements to a list is easy:

```
void add_point(Point& p, list<Point>::iterator i)
{
  pts.push_front(p);        //add at beginning
  pts.push_back(p);         //add at end
  pts.insert(i,p);          //add before the element 'i' refers to
}
```

**Map**

Writing code to look up the coordinates of a point in a list of points is really quite tedious. In addition, a linear search is quite inefficient for all but the shortest lists. (It's more efficient to use a tree–like structure to do a search. It's how you look up a name in the phone book. You open the book near the midpoint and go to the earlier or later pages depending on whether what your want is before or after where you are at in the phone book.) Other data structures directly support insertion, deletion, and searching based on values. In particular, the standard library provides the `map` type. A `map` is a container of pairs of values. For example:

```
#include <string>
#include <map>

class Point;
map<string,Point> California;   // A city name paired with a point
```

Here we might imagine a map of California with points corresponding to cities. The name of a city is a string. In other contexts, a `map` is known as an associative array or a dictionary.

When indexed by a value of its first type (called the *key*) a `map` returns the corresponding value of the second type (called the *value* or the *mapped type*). For example:

```
void print_city(const string& s)
  {
    if (Point i = California[s])
      cout << s << ' ' << i.x() << " , " << i.y() << endl;
  }
```

If no match is found for the key `s`, a default value is returned from the map object. For example, the default value for `California` could be 0. Then `Point i` is zero and the if statement is false.

Each element in a map is a `pair`. The first element of a `pair` is called `first`, and the second element is called `second`. For example, suppose we have a `map` that consists of a sequence of (string,int) pairs. Then we could define a `print` function as follows:

```
#include <algo>
#include <string>
#include <map>
#include <iostream.h>

void print(const pair<const string,int>& r)
  { cout << r.first << r.second << '\n'; }

int main()
```

```
{
  map<string,int> histogram;
  //initialize pairs...

  for_each(histogram.begin(),histogram.end(),print);
}
```

where `for_each` is an algorithm that applies the `print` function to every element of a sequence, in this case, to every `pair` of a map.

The elements of a map are sorted in a particular order so that the less–than operation is defined for its key types. For elements for which there is no obvious order or when there is no need to keep the container sorted, one might consider using a hash map.

**Comments on Standard Containers**

A `map`, a `list`, and a `vector` can each be used to represent a collection of objects like `pts`. However, each has strengths and weaknesses. For example, subscripting a `vector` is cheap and easy. On the other hand, inserting an element between two elements tends to be expensive. A `list` has exactly the opposite properties; it's hard to subscript but it's easy to insert an element anywhere in the list. A `map` resembles a `list` of (key, value) pairs except that it is optimized for finding values based on keys.

The standard containers and their basic operations are designed to be similar from a notational point of view. Furthermore, the meanings of the operations are equivalent for the various containers. For example, `push_back()` can be used (reasonably efficiently) to add elements to the end of a `vector` as well as for a `list`, and every container has a `size()` member function that returns its number of elements.

**Algorithms**

A data structure, such as a list or a vector, is not very useful on its own. To use one, we need operations for basic access such as adding and removing elements. Furthermore, we rarely just store objects in a container. Consequently, the standard library provides the most common algorithms for containers in addition to providing the most common container types. The algorithms are expressed in terms of sequences of elements. A sequence is represented by a pair of iterators specifying the first element and the one–beyond–the–last element. For example, the following sorts a vector and places a copy of each unique `vector` element in a `list`:

```
#include <list>
#include <vector>
#include <algo>
void f(vector<float>& ve, list<float>& le)
 {
   sort(ve.begin(),ve.end());
   unique_copy(ve.begin(),ve.end(),le.begin());
 }
```

In the example, `sort()` sorts the sequence in increasing order from `ve.begin()` to `ve.end()` - which just happens to be all the elements of a `vector`. Of course, sorting only makes sense if having one element is greater than (>) another is defined. For writing, you need only specify the first element to be written. If more than one element is written, the elements following that initial element will be overwritten. If we want to add the new elements to the end of a container, we can write:

```
#include <list>
#include <vector>
#include <algo>

void f(vector<float>& ve, list<float>& le)
 {
   sort(ve.begin(),ve.end());
   unique_copy(ve.begin(),ve.end(),le.back_inserter(le));
 }
```

When you first encounter a container, a few iterators referring to useful elements can be obtained; `begin()` and `end()` are the best examples of this. In addition, many algorithms return iterators. For example, the standard algorithm `find` looks for a value in a sequence and returns an iterator to the element found.

```
#include <string>
#include <iostream.h>
#include <algo>

void find_it(const string& s, char c)
 {
   string::const_iterator i = find(s.begin(), s.end(), c);
   if(i != s.end())
     cout << "found " << c << endl;
 }
```

The `find` algorithm returns an iterator to the first occurrence of a value in a sequence or the one–past–the–end iterator.

Counting occurrences of an element is another algorithm provided by the standard library. `count` takes a sequence as its argument, rather than a container. For example,

```
#include <string>
#include <iostream.h>
#include <algo>
#include <complex>

  void f(list<complex>& lc, vector<string>& vc, string s);
```

```
{
  int i1 = count(lc.begin(),lc.end(),complex(1,3));
  int i2 = count(vc.begin(),vc.end(),"Michael");
  int i3 = count(s.begin(), s.end(), 'x');
}
```

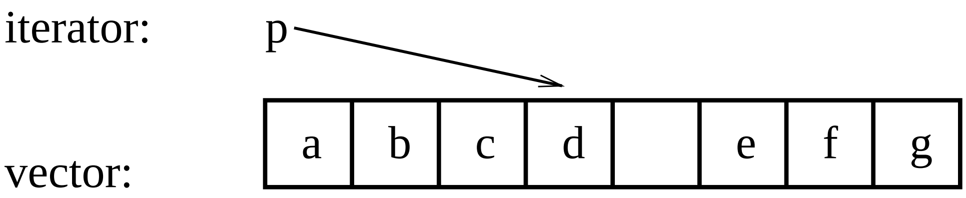
Some algorithms apply a function to the elements of a sequence. `for_each` is an example of this. It goes through the elements of a sequence and applies a function to each element. For example, suppose that we have the class `Student` and that each student has a name. Let `print` be a function that prints the name of a student.

```
#include <iostream.h>
#include <algo>
#include <list>

 class Student;
 void print(Student& st)
 {
   cout << st.name() << endl;
 }

 main()
 {
   list<Student> roll;
    // initialize students ...
   // print names of all the students in roll

      for_each(roll.begin(),roll.end(),print);
 }
```

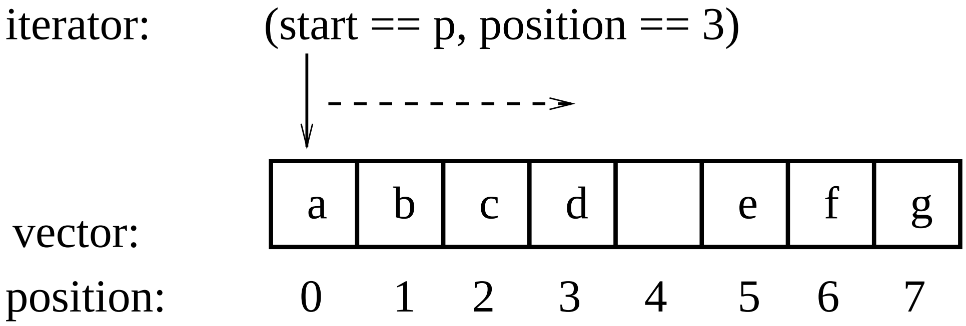
`for_each` goes through the `list` of students and prints the name of each student. Notice that we pass do not pass the container `list`, but rather a sequence by indicating the beginning and end of the sequence.

**Iterators**

What are iterators really? Any particular iterator is an object of some type. There are, however, many different iterator types because an iterator needs to hold the information necessary for doing its job for a particular container type. These iterator types can be as different as the containers and the specialized needs they serve. For example, a `vector`'s iterator is most likely an ordinary pointer because a pointer is quite a reasonable way of referring to an element of a `vector`:

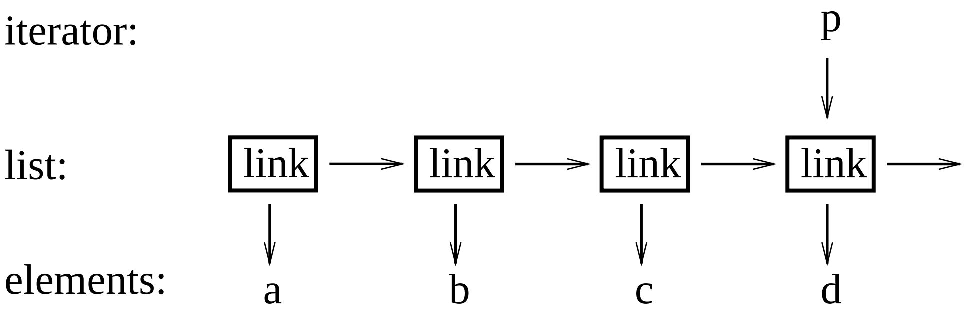



Alternatively, a `vector` iterator could be implemented as a pointer to the `vector` plus an index:




Using such an iterator would allow bounds checking.

A list iterator must be something more complicated than a simple pointer to an element because an element of a list in general does not know where the next element of that list is. Thus, a list iterator might be a pointer to a link:




What is common for all iterators is their semantics and the naming of their operations. For example, applying `++` to any iterator yields an iterator that refers to the next element. Similarly, `*` yields the element to which the iterator refers. In fact, any object that obeys a few simple rules like these is an iterator. Furthermore, users rarely need to know the type of a specific iterator; each container "knows" its iterator types and makes them available under the conventional names `iterator` and `const_iterator`. (We use `const_iterator` if we are not going to change the element.) For example, `list<Point>::iterator` is the general iterator type for `list<Point>`. We rarely have to worry about the details of how that type of iterator is defined.