Chapter 20
The STL
(containers, iterators, and algorithms)

Bjarne Stroustrup
www.stroustrup.com/Programming

Abstract
• This lecture and the next present the STL – the containers and algorithms part of the C++ standard library
• The STL is an extensible framework dealing with data in a C++ program.
• First, I will present the general ideal, then the fundamental concepts, and finally examples of containers and algorithms.
• The key notions of sequence and iterator used to tie data together with algorithms (for general processing) are also presented.

Overview
• Common tasks and ideals
• Generic programming
• Containers, algorithms, and iterators
• The simplest algorithms: find()
• Parameterization of algorithms
  – std::find and function objects
• Sequence containers
  – vector and list
• Associative containers
  – map, set
• Standard algorithms
  – copy, sort, ...
  – Input iterators and output iterators
• List of useful facilities
  – Headers, algorithms, containers, function objects

Common tasks
• Collect data into containers
• Organize data
  – For printing
  – For fast access
• Retrieve data items
  – By index (e.g., get the Nth element)
  – By value (e.g., get the first element with the value "Chocolate")
  – By properties (e.g., get the first elements where "age<64")
• Add data
• Remove data
• Sorting and searching
• Simple numeric operations

Observation
We can (already) write programs that are very similar independent of the data type used
  – Using an int isn’t that different from using a double
  – Using a vector<int> isn’t that different from using a vector<string>

Ideals
We’d like to write common programming tasks so that we don’t have to re-do the work each time we find a new way of storing the data or a slightly different way of interpreting the data
  – Finding a value in a vector isn’t all that different from finding a value in a list or an array
  – Looking for a string ignoring case isn’t all that different from looking at a string not ignoring case
  – Graphing experimental data with exact values isn’t all that different from graphing data with rounded values
  – Copying a file isn’t all that different from copying a vector
Ideals (continued)

• Code that’s
  – Easy to read
  – Easy to modify
  – Regular
  – Short
  – Fast
• Uniform access to data
  – Independently of how it is stored
  – Independently of its type
• ...

Examples

• Sort a vector of strings
• Find an number in a phone book, given a name
• Find the highest temperature
• Find all values larger than 800
• Sort the telemetry records by unit number
• Sort the telemetry record by time stamp
• Find the first value larger than “Petersen”?  
  • What is the largest amount seen?
  • Find the first difference between two sequences
• Compute the pair wise product of the elements of two sequences
• What’s the highest temperatures for each day in a month?
• What’s the top 10 best-sellers?
• What’s the entry for “C++” (say, in Google)?
• What’s the sum of the elements?
• ...

Generic programming

• Generalize algorithms
  – Sometimes called “lifting an algorithm”
• The aim (for the end user) is
  – Increased correctness
    • Through better specification
    • Greater range of uses
    • Possibilities for re-use
  – Better performance
    • Through wider use of tuned libraries
    • Unnecessarily slow code will eventually be thrown away
• Go from the concrete to the more abstract
  – The other way most often leads to bloat

Lifting example (concrete algorithms)

```c
double sum(double array[], int n) // one concrete algorithm (doubles in array)
{
    double s = 0;
    for (int i = 0; i < n; ++i) s = s + array[i];
    return s;
}

struct Node { Node* next; int data; };
int sum(Node* first) // another concrete algorithm (ints in list)
{
    int s = 0;
    while (first) {
        s += first->data;
        first = first->next;
    }
    return s;
}
```

Lifting example (abstract the data structure)

```c
// pseudo-code for a more general version of both algorithms
int sum(data) // somehow parameterize with the data structure
{
    int s = 0; // initialize
    while (not at end) { // loop through all elements
        s += get value; // compute sum
        get next data element;
    }
    return s; // return result
}
```

• We need three operations (on the data structure):
  – not at end
  – get value
  – get next data element
Lifting example (STL version)

```cpp
// Concrete STL-style code for a more general version of both algorithms
template<class Iter, class T>
// Iter should be an Input_iterator
// T should be something we can + and =
T sum(Iter first, Iter last, T s)
{
  while (first!=last) {
    s = s + *first;
    ++first;
  }
  return s;
}
• Let the user initialize the accumulator
  float a[] = { 1,2,3,4,5,6,7,8 };
  double d = 0;
  d = sum(a,a+sizeof(a)/sizeof(*a),d);
```

Lifting example

• Almost the standard library accumulate
  – I simplified a bit for terseness
  (see 21.5 for more generality and more details)
• Works for
  – arrays
  – vectors
  – lists
  – istreams
  – …
• Runs as fast as “hand-crafted” code
  – Given decent inlining
• The code’s requirements on its data has become explicit
  – We understand the code better

The STL

- Part of the ISO C++ Standard Library
- Mostly non-numerical
  - Only 4 standard algorithms specifically do computation
    - Accumulate, inner_product, partial_sum, adjacent_difference
  - Handles textual data as well as numeric data
  - E.g. string
  - Deals with organization of code and data
  - Built in types, user-defined types, and data structures
- Optimizing disk access was among its original uses
- Performance was always a key concern

The STL

- Designed by Alex Stepanov
- General aim: The most general, most efficient, most flexible representation of concepts (ideas, algorithms)
  - Represent separate concepts separately in code
  - Combine concepts freely wherever meaningful
- General aim to make programming “like math”
  – or even “Good programming is math”
  – works for integers, for floating-point numbers, for polynomials, for …

Basic model

- Algorithms
  - sort, find, search, copy, …
- Separation of concerns:
  - Algorithms manipulate data, but don’t know about containers
  - Containers store data, but don’t know about algorithms
  - Algorithms and containers interact through iterators
  - Each container has its own iterator types
- Containers
  - vector, list, map, hash_map, …

The STL

- An ISO C++ standard framework of about 10 containers and about 60 algorithms connected by iterators
  – Other organizations provide more containers and algorithms in the style of the STL
  - Boost.org, Microsoft, SGI, …
- Probably the currently best known and most widely used example of generic programming
The STL

- If you know the basic concepts and a few examples you can use the rest
- Documentation
  - SGI
    - http://www.sgi.com/tech/stl/ (recommended because of clarity)
  - Dinkumware
    - http://www.dinkumware.com/refxcpp.html (be aware of several library versions)
  - Rogue Wave
  - More accessible and less complete documentation
    - Appendix B

Basic model

- A pair of iterators define a sequence
  - The beginning (points to the first element — if any)
  - The end (points to the one-beyond-the-last element)

An iterator is a type that supports the “iterator operations”
- ++ Go to next element
- * Get value
- == Does this iterator point to the same element as that iterator?
- Some iterators support more operations (e.g., ~, +, and [ ])

Containers

- vector
  - (hold sequences in difference ways)
- list
  - (doubly linked)
- set
  - (a kind of tree)

The simplest algorithm: find()

```
find(In first, In last, const T& val)

// Find the first element that equals a value
begin: end:
```

void f(vector<int>& v, int x)

```
vector<int>::iterator p = find(v.begin(), v.end(), x);
if (p != v.end()) { /* we found x */ } // ...
```

void f(list<string>& v, string x)

```
list<string>::iterator p = find(v.begin(), v.end(), x);
if (p != v.end()) { /* we found x */ } // ...
```

void f(set<double>& v, double x)

```
set<double>::iterator p = find(v.begin(), v.end(), x);
if (p != v.end()) { /* we found x */ } // ...
```

Algorithms and iterators

- An iterator points to (refers to, denotes) an element of a sequence
- The end of the sequence is “one past the last element”
  - not “the last element”
  - That’s necessary to elegantly represent an empty sequence
- One-past-the-last-element isn’t an element
  - You can compare an iterator pointing to it
  - You can’t dereference it (read its value)
- Returning the end of the sequence is the standard idiom for “not found” or “unsuccessful”
Simple algorithm: `find_if()`

- Find the first element that match a criterion (predicate)
  - Here, a predicate takes one argument and returns a `bool`

```cpp
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
    while (first!=last && !pred(*first)) ++first;
    return first;
}
```

```cpp
void f(vector<int>& v)
{
    vector<int>::iterator p = find_if(v.begin(),v.end,Odd());
    if (p!=v.end()) { /* we found an odd number */ }
    // …
}
```

Predicates

- A predicate (of one argument) is a function or a function object that takes an argument and returns a `bool`
- For example
  - A function
    ```cpp
    bool odd(int i) { return i%2; } // Is %2 odd?
    ```
  - A function object
    ```cpp
    struct Odd {
        bool operator()(int i) const { return i%2; }
    };
    Odd odd; // Make an object odd of type Odd
    odd?; // Is odd: is i%2 odd?
    ```

Function objects

- A concrete example using state

```cpp
template<T> struct Less_than {
    T val;
    bool operator()(const T& x) const { return x < val; }
};
```

```cpp
// find x<43 in vector<int>:
p=find_if(v.begin(), v.end(), Less_than(43));
```

```cpp
// find x<"perfection" in list<string>:
q=find_if(ls.begin(), ls.end(), Less_than("perfection"));
```

Policy parameterization

- Whenever you have a useful algorithm, you eventually want to parameterize it by a “policy”.
  - For example, we need to parameterize sort by the comparison criteria

```cpp
struct Cmp_by_name {
    bool operator()(const Rec& a, const Rec& b) const
    { return a.name < b.name; } // Look at the name field of Rec
};
```

```cpp
struct Cmp_by_addr {
    bool operator()(const Rec& a, const Rec& b) const
    { return 0 < strncmp(a.addr, b.addr, 24); } // Correct?
};
```

Comparisons

- Different comparisons for `Rec` objects

```cpp
  // Different comparisons for Rec objects:
  struct Cmp_by_name {
      bool operator()(const Rec& a, const Rec& b) const
      { return a.name < b.name; } // Look at the name field of Rec
  };
  struct Cmp_by_addr {
      bool operator()(const Rec& a, const Rec& b) const
      { return 0 < strncmp(a.addr, b.addr, 24); } // Correct?
  };
```

- Note how the comparison function objects are used to hide ugly error-prone code
template<class T> class vector {
    T* elements;
    // ...
    typedef ??? iterator;  // the type of an iterator is implementation defined
    // and it (usefully) varies (e.g. range checked iterators)
    // a vector iterator could be a pointer to an element
    typedef ??? const_iterator;
    iterator begin();  // points to first element
    const_iterator begin() const;
    iterator end();    // points one beyond the last element
    const_iterator end() const;
    iterator erase(iterator p);   // remove element pointed to by p
    iterator insert(iterator p, const T& v);  // insert a new element v before p
};

insert() into vector

vector<int>::iterator p = v.begin(); ++p; ++p; ++p;
vector<int>::iterator q = p; ++q;

v: 6 7 8 9 0 1 2 3 4 5
p: 
q: 

p = v.insert(p,99); // leaves p pointing at the inserted element

v: 6 7 8 9 0 1 2 3 4 9 9 5
p: 
q: 

Note: q is invalid after the insert()
Note: Some elements moved, all elements could have moved

erase() from vector

v: 7
p: 
q: 

p = v.erase(p);  // leaves p pointing at the element after the erased one

v: 6 0 1 2 3 4 5
p: 
q: 

• vector elements move when you insert() or erase()
• Iterators into a vector are invalidated by insert() and erase().

list

template<class T> class list {
    Link* elements;
    // ...
    typedef ??? iterator;  // the type of an iterator is implementation defined
    // and it (usefully) varies (e.g. range checked iterators)
    // a list iterator could be a pointer to a link node
    typedef ??? const_iterator;
    iterator begin();   // points to first element
    const_iterator begin() const;
    iterator end();     // points one beyond the last element
    const_iterator end() const;
    iterator erase(iterator p);  // remove element pointed to by p
    iterator insert(iterator p, const T& v); // insert a new element v before p
};

insert() into list

list<int>::iterator p = v.begin(); ++p; ++p; ++p;
list<int>::iterator q = p; ++q;

v: 6 0 1 2 3 4 5
p: 
q: 

v = v.insert(p,99);  // leaves p pointing at the inserted element

v: 6 0 1 2 3 4 5 9 9
p: 
q: 

• Note: q is unaffected
• Note: No elements moved around

erase() from list

v: 7
p: 
q: 

p = v.erase(p);  // leaves p pointing at the element after the erased one

v: 6 0 1 2 3 4 5
p: 
q: 

• Note: list elements do not move when you insert() or erase()
Ways of traversing a vector

```cpp
for(int i = 0; i<v.size(); ++i) // why int?
    // do something with v[i]
for(vector<int>::size_type i = 0; i<v.size(); ++i) // longer but always correct
    // do something with v[i]
for(vector<int>::iterator p = v.begin(); p!=v.end(); ++p)
    // do something with *p
```

- Know both ways (iterator and subscript)
  - The subscript style is used in essentially every language
  - The iterator style is used in C (pointers only) and C++
  - The iterator style is used for standard library algorithms
  - The subscript style doesn’t work for lists (in C++ and in most languages)
- Use either way for vectors
  - There are no fundamental advantages of one style over the other
  - But the iterator style works for all sequences
  - Prefer `size_type` over plain `int`
    - Pedantic, but quells compiler and prevents rare errors

Some useful standard headers

- `<iostream>` I/O streams, cout, cin, …
- `<fstream>` file streams
- `<algorithm>` sort, copy, …
- `<numeric>` accumulate, inner_product, …
- `<functional>` function objects
- `<vector>`
- `<map>`
- `<list>`
- `<set>`

Next lecture

- Map, set, and algorithms