Teaching with the STL

Joseph Bergin
Pace University
Michael Berman
Rowan College of New Jersey

Part 1

Introduction to STL Concepts

STL: What and Why

- Generic data structures (containers) and algorithms for operating upon them
- Part of the C++ definition
- Developed by Stepanov, Lee, and Musser
- Generic programming != OOP (no encapsulation)
- Extensible -- you can add your own elements

Templates are not Classes

- These are not cookies
- You can't eat them
- They can be used to make cookies

Templates are not Classes

- These are cookies
- They are made with a cookie cutter
- You can eat them

Templates are not Cookies

- Templates are used to create classes
- You can't compile them
- You can instantiate them
  - This gives you a class
- The instantiations are compiled
- The instantiations are strongly typed like other classes

Templates are not Classes

```
template <class E>
class stack                                   <- A class template
{       ...
        void push(E e){...}
}


stack <int> S;                                <- a template class

S.push(55);
```

Templates are not Functions

```
template <class E>              <- a function template
E& min(E& a, E& b)
{       if(a < b) return a;
        return b;
}


abox = min(box1, box2);         <- a template function
```

The Standard Template Library

- Containers
  - array, vector, deque, list, set, map, multiset, multimap
- Algorithms
  - sort, search, and nearly everything else
- Iterators
  - generalize pointers and pointer arithmetic
- Adaptors
  - change the behavior of other components
- Allocators
  - memory management

The Standard Template Library

- Containers
  - array, vector, deque, list, set, map, multiset, multimap
- Algorithms
  - sort, search, and nearly everything else
- Iterators
  - generalize pointers and pointer arithmetic
- Adaptors
  - change the behavior of other components
- Allocators
  - memory management

The Major Dimensions

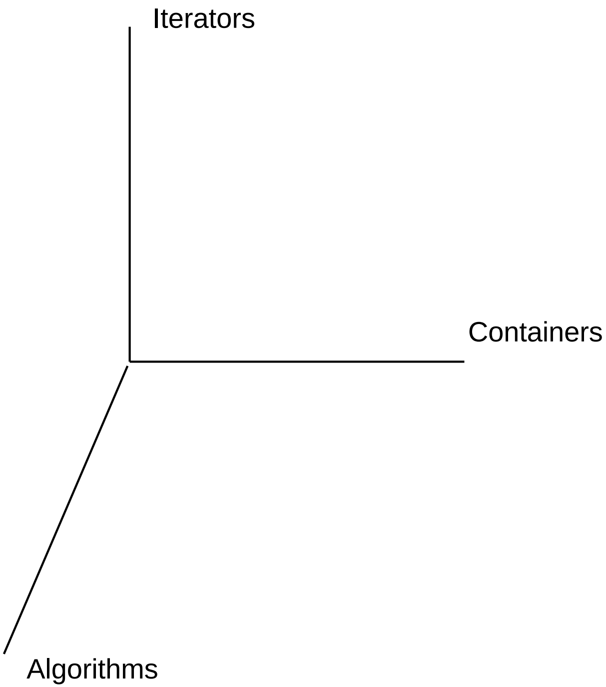
Independent Development of:

- Containers
  - contain values
- Algorithms
  - operate on containers
- Iterators
  - interface between containers and algorithms

The Major Dimensions

Independent Development of:

- Containers
  - contain values
- Algorithms
  - operate on containers
- Iterators
  - interface between containers and algorithms



Simple Examples

```
// Summing a list -- typical early assignment
int main()
{
    int sum(0);
    cout << "Enter your integers:\n";
    while(1) {
            int entry;
            cin >> entry;
            if (cin.eof()) break;
            sum += entry;
    }
    cout << "The sum was " << sum << endl;
    return 0;
}
```

… just add STL!

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

int main()
{
    cout << "Enter your integers:" << endl;
    // set up an input stream iterator for cin
    istream_iterator < int, ptrdiff_t > cinIter(cin), eos;
    // accumulate the sum using the input iterator
    int sum = accumulate(cinIter, eos, 0);
    cout << "The sum was " << sum << endl;
    return 0;
}
```

Vector Manipulation

```
vector<int> v;
v.push_back(3);
v.push_back(4);
v.push_back(5);
v.push_back(6);

vector<int>::iterator i;

for(i = v.begin(); i != v.end(); ++i)
        cout << *i << endl;
sort(v.begin(); v.end());
for(i = v.begin(); i != v.end(); ++i)
        cout << *i << endl;
```

Sorting a file of strings

```
cout << "Enter name of file to sort: ";
string sortFileName;
cin >> sortFileName;
ifstream sortFile(sortFileName.c_str());
istream_iterator < string, ptrdiff_t >
      sortFileIter(sortFile), eos;
vector < string > sortVector;
copy(sortFileIter, eos, back_inserter(sortVector));
cout << "Sorting " << sortVector.size() << " words.\n";
sort(sortVector.begin(), sortVector.end());
ostream_iterator < string > coutIter(cout, "\n");
cout << "Sorted file is: \n";
copy(sortVector.begin(), sortVector.end(), coutIter);
```

Iterators are the Key

Containers +
Iterators +
Algorithms
= STL Programs

Iterator Flavors

- Forward Iterators (operator++)
  - Input Iterators
  - Output Iterators
- Bidirectional Iterators (operator --)
- Random Access Iterators (operator +=)

Iterator Flavors

- Forward Iterators (operator++)
  - Input Iterators
  - Output Iterators
- Bidirectional Iterators (operator --)
- Random Access Iterators (operator +=)

All Iterators have operator*
All Containers produce iterators begin() and end()
begin references first. end is “after” last

Slouching Toward Iterators

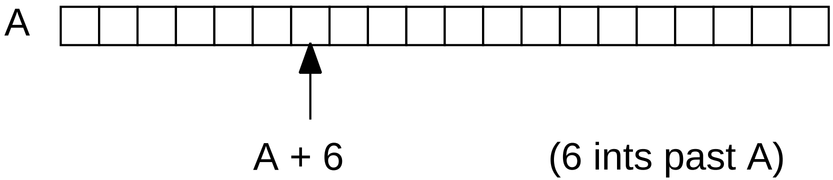
```
template < class T >
void selectionSort(T   elements[ ]  , int length)
{          for(int i = 0; i < length - 1; ++i)
           {          int s = i;
                      T small =  elements[s]  ;
                      for(unsigned j = i + 1; j < length; ++j)
                                 if( elements[j]   < small)
                                 {          s = j;
                                            small =  elements[s]  ;
                                 }
                      elements[s]   =  elements[i]  ;
                      elements[i]   = small;
           }
}
```

Pt. 1: Dependent on Arrays

Pointer Duality Law

int * A = new int [20];

A[i] is equivalent to ***(A + i)**



Slouching Towards Iterators

```
int elements [20] = ...
selectionSort(elements, 20)
```

```
int * start = elements;
int * end = elements + 20; // or &elements[20]

selectionSort(start, end);
```

Pt. 2: The Goal

The Replacements

```
template < class T >
void selectionSort(T    elements[ ]  , int length)
{         for(int i = 0; i < length - 1; ++i)
          {         int s = i;
                    T small =  elements[s]  ;
                    for(unsigned j = i + 1; j < length; ++j)
                              if( elements[j]   < small)
                              {         s = j;
                                        small =  elements[s]  ;
                              }
                    elements[s]   = elements[i]  ;
                    elements[i]   = small;
          }
}
```

start = elements
end = elements + length
loc = & elements[s]
where = & elements[i]
inner = & elements[j]

Slouching Towards Iterators

```
template < class T >
void selectionSort(T* start, T* end)
{       for(T* where = start ; where < end - 1 ; ++where)
        {       T* loc = where;
                T small = *loc;
                for(T* inner = where + 1; inner < end; ++inner)
                        if(*inner < *loc)
                        {       loc = inner;
                                small = *loc;
                        }
                *loc = *where;
                *where = small;
        }
}
```

Pt 3: The Result (almost)

Slouching Towards Iterators

```
template < class Iterator , class value_type>
void selectionSort_aux(Iterator start, Iterator end, value_type)
{       for(Iterator where = start ; where < end - 1 ; ++where)
        {       Iterator loc = where;
                value_type small = *loc;
                for(Iterator inner = where + 1; inner < end;
                                ++inner)
                        if(*inner < *loc)
                        {       loc = inner;
                                small = *loc;
                        }
                *loc = *where;
                *where = small;
        }
}
```

Pt 3: The Result (…)

Slouching Towards Iterators

```
template < class Iterator >
inline void selectionSort (Iterator start, Iterator end)
{
        selectionSort_aux(start, end, *start);
}
```

Pt 3: The Result !

The Advantages

- This version will sort more than arrays.
  - All we need is a structure referenced by a datatype like a pointer that implements
    - operator *
    - operator++
    - operator+
    - operator-
    - operator=
    - operator<

With care we could reduce this list

Such datatypes are called iterators

The Lesson

- Implement containers separate from algorithms
- Use pointer-like structures as an interfacing mechanism

The Lesson

- Implement containers separate from algorithms
- Use pointer-like structures as an interfacing mechanism

To Gain →

Advantages

- Generality
- A framework for thinking about containers and algorithms
- Smaller written code
- Smaller compiled code

Advantages

- Generality
- A framework for thinking about containers and algorithms
- Smaller written code
- Smaller compiled code

But... →

Disadvantages

- Students must become thoroughly familiar with all aspects of pointers including
  - The pointer duality law
  - Pointer arithmetic
  - Pointer “gotchas”

Iterators, Pro and Con

- Iterators make it possible for the STL to be efficient and flexible, but…
- As an iterator-based library, the STL provides *no safety* for the user
- Basically, if you like C++, you will probably like the STL!

Exercise

Rewrite this search function using Iterators:

```
template <class T>
int linearSearch(T a[], int length, T target)
{
    for (int i = 0; i < length; i++)
        if (target == a[i])
            return i; // return position in array
    return -1; //use -1 to indicate item not found
}
```

Sample solution

```
template <class Iterator, class T>
Iterator linearSearch(Iterator begin, Iterator end,
                      T target)
{
          for (Iterator i = begin; i < end; i++)
                    if (target == *i)
                              return i;
          return end; //use end to indicate item not found
}

....
int a[] = {1, 2, 3, 4, 5};
if (int * pos = linearSearch(a, a+5, 4) < a+5)
          cout << "found " << *p << endl;
else
          cout << "not found\n";
```

Iterators for STL Containers

- Work like a pointer
- To access an individual member of a container, you dereference an iterator (*)
- Advance iterator with ++
- Special past-the-end iterator used to mark end of container (improper to dereference)
- STL typically does not bounds check iterators
- Containers provide an iterator constructor

Extended Example: Using Vectors

- Vectors are a generic container provided by STL
- Similar to built-in arrays, but grow dynamically at the end

Vector example, p. 1

```
#include <iostream.h>
#include <vector.h>
#include <algo.h>
template <class T>
void vectorPrint(char * label, vector<T> v)
{
    cout << label << endl;
    for (int i=0; i < v.size(); i++)
        cout << v[i] << '\t';
    cout << "\n\n";
}
```

Vector example, part 2

```
int main()
{
	// create a vector containing ints from 1 to 10
	vector<int> v1;
	for (int i = 0; i < 10; i++)
		v1.push_back(i+1);
	vectorPrint("v1 initial state", v1);

	// create a second vector of the same size
	vector<int> v2(v1.size());

	// copy vector v1 to v2
	copy(v1.begin(), v1.end(), v2.begin());
	vectorPrint("v2 should be a copy of v1", v2);
```

Vector example, p.3

```
cout << "v1 == v2? " << (v1 == v2) << endl;

// create another vector and fill it
vector<double> v3(20);
fill(v3.begin(), v3.begin() + 20, 3.14);
vectorPrint("vector v3", v3);

// rotate the first vector
rotate(v1.begin(), v1.begin()+5, v1.end());
vectorPrint("v1 rotated", v1);

// now, sort it
sort(v1.begin(), v1.end());
vectorPrint("v1 sorted", v1);
```

Vector example, p. 4

```
// find the location of 5 in v2
vector<int>::iterator loc;
loc = find(v1.begin(), v1.end(), 5);
cout << "this should be 5: " << *loc << endl;

}
```

Exercise

Write a program that reads in an arbitrary number of double-precision floating point numbers > 0.0, terminated by 0.0, puts them into a vector, sorts them, and finds the median.

Sample solution

```
int main()
{       vector<double> v;
        double x;
        cin >> x;
        while (x > 0.0)
        {       v.push_back(x);
                cin >> x;
        }
        sort(v.begin(), v.end());
        if (v.size() % 2 == 0) // even case
                cout << (v[v.size()/2 - 1] +
                        v[(v.size()/2)])/2 << endl;
        else
                cout << v[v.size()/2] << endl;

}
```

Five classes of iterators

Iterators classified by the operations they support

- input
  - can be compared (==, !=), incremented, dereferenced as rvalue (assign from)
  - usually used with input streams (cin)
  - single pass
- output
  - compared, dereferenced as lvalue (assign to)
  - usually used with output streams (cout)
  - single pass

Iterator Classes

- forward
  - union of features of input/output iterators
  - plus, can be traversed more than once
- bidirectional
  - does everything a forward iterator does
  - plus, can traverse in reverse, using “--”

Iterator Classes

- random access
  - » does everything a bidirectional iterator does
  - » plus can access any location in constant time
  - » supports pointer-like arithmetic, e.g. **i + 7**
  - » can compare relationally: **<, >, <=, >=**

Relationship among algorithms, iterators, containers

- Iterators form a hierarchy

- Algorithms classified by the iterators they *require*; containers by the iterators they *support*
- If an algorithm requires a particular iterator, it can also use those higher in the hierarchy

Sample Relationships

- Lists provide bidirectional iterators (*not* random access) so you can’t use the sort algorithm, nor can you use a list as a heap. (But lists have a special sort method.)
- The next_permutation algorithm requires a bidirectional iterator, so you can use it on a list (or a vector…) but not on the input stream.

Containers

- Objects that store other objects
- All STL containers are generic (templated) but homogeneous
- Built-in C-style arrays work as containers
- Sequence Containers: *vector, deque, list*
- Associative Containers: *set, multiset, map, multimap*

Containers

- Ordinary Arrays (I.e. regions of memory)
- Vectors -- expandable array
- Deques -- expandable at both ends
- Lists -- doubly linked circular with header
- Sets and Multisets -- red-black tree
- Maps and Multimaps -- dictionary like

Note: Implementation is not specified
but efficiency is specified.

All Containers Provide

- A Storage Service
  - insert and erase...
- An Associated Iterator type
  - The type of iterator determines what can be done with the container.
- begin() and end() iterators - - - [b, e)
- A collection of types: vector::value_type...
- constructors, assignment, cast, equality...

Sequence Containers

- vector
  - O(1) access to any element, O(1) (amortized) to add to end, O(n) to add elsewhere
  - grows dynamically as objects added to end; vector handles storage management
  - use [ ] syntax to access elements
  - use **push_back()** to add to end of vector (and grow size), **insert()** to add within (also **pop_back()**
  - Fastest (average) container for most purposes.

Vector Example

```
vector < int > v;
v.push_back(47);
v.push_back(17);
cout << v.size() << ‘\t’ <<
    v[0] << ‘\t’ << v[1] <<
    endl;
```

```
2   47   17
```

Sequence Containers

- deque
  - Expandable “array” at both ends
  - push_front, pop_front
  - Average O(1) insert at both ends
  - Linear insert in middle
  - Random Access Iterators
  - Good choice for queues & such.

Deque Example

```
deque < char > dc;
dc.push_back('h');
dc.push_back('o');
dc.push_front('i');
dc.push_front('h');
cout << dc.size() << ‘\t’ <<
dc[0] << dc[1] << dc[2] << dc[3]
    endl;
2   hiho
```

Sequence Containers

- list
  - » doubly-linked list
  - » O(1) insertions, *no random access*
  - » Slower on average than vector or deque
  - » special functions for splicing, merging, etc.
  - » can use **push_front** , **push_back** , and **insert**
  - » access items using bidirectional iterators

Associative Containers: sets and multisets

- two template parameters: a key type and a comparison relation (sets are ordered)
- insert and find operations are O(log n)
- so really, a set is a balanced binary tree that stores just keys (no associated data)
- only difference between set and multiset is duplicate keys in multiset

Set example

```
set < int, greater < int  > > intSet;
intSet.insert(2);
intSet.insert(2);
intSet.insert(7);
cout << intSet.count(2) << '\t' <<
                          intSet.count(17) << endl;
```

```
      1  0
```

Associative Containers: Maps and Multimaps

- Just like a set, except key/data pairs
- e.g. a symbol table, a dictionary
- Performance is tree-like, i.e. O(log n)
- Can retrieve the members in order, using iterators
- Uses the parameterized type **pair < keyType, dataType >**

Iterator classes for STL containers

- random access: vector, deque, C array
- bidirectional: list, set, multiset, map, multimap
- input: istream, const C array
- output: ostream

Algorithms

- Defined in terms of a specific iterator type
  - e.g. sort requires random access iterators
- Work with all containers that provide that iterator type -- including user written.
- Combine good generality w/ good efficiency
- Do not appear within container classes (generally)
  - This is important to generality & efficiency

Classification of Algorithms

- Non-mutating sequence algorithms
  - do not modify contents of container
  - most require a pair of iterators, specifying a range
  - e.g.: find, for_each, count
- Mutating sequence algorithms
  - modify contents
  - e.g.: copy, fill, reverse, rotate

Example: find

```
int a[] = {1,2,3,14,4};
vector<int> v(a, a+5); // construct vector from array
vector<int>::iterator v_loc;
v_loc = find(v.begin(), v.end(), 3);
// v_loc is now an iterator pointing to 3 in v

list<int> l(a, a+5); // construct a list from array
list<int>::iterator list_loc;
list_loc = find(l.begin(), l.end(), 4);
// list_loc is an iterator pointing to 4 in l
list_loc = find(l.begin(), l.end(), -1);
// list_loc is an iterator pointing to l.end()
// to indicate failure
```

Example: for_each

```
void print_char(char c)
{
        cout << c << endl;
}
int main()
{
        char s[] = "abcdefg";
        vector<char> vc(s, s + strlen(s));
        set<char> sc(s, s + strlen(s));
        for_each(vc.begin(), vc.end(), print_char);
        for_each(s, s+strlen(s), print_char);
        for_each(sc.begin(), sc.end(), print_char);
}
```

Classification of Algorithms

- Sorting and related
  - sort, nth_element, binary_search, min/max
- Numeric
  - accumulate, inner_product, partial_sum

Function Objects 1

- Predicates
  - » A function of one argument returning bool
- Comparisons
  - » A function of two arguments returning bool
- Unary Operator, Binary Operator
  - » A function of one or two arguments returning a value

Function Objects 2

- Can be functions or template functions
- Can be objects implementing an appropriate operator()
- Many are built in
  - less..., plus..., and...,...
- Function adaptors too
  - not1, not2, bind1st, bind2nd,...

Function Object Example

```
class stringLess
{          bool operator()(char* s1, char* s2)
           {          return strcmp(s1, s2) < 0;
           }
           . . .
} // Defines a function object.

vector< char* > stringVec;
. . .
sort (stringVec.begin(), stringVec.end(), stringLess());
// Note the constructor call in the last argument ^^^^
```

Adaptors

- Change the interface of a component
- Example: stack adaptor lets you use a list or deque with push and pop
- insert iterator adaptors: used to grow a container. Most common is back_inserter()
- istream_iterator, ostream_iterator let you use input and output streams as containers

Stack/Queue example (part 1)

```
typedef list < int > intList;
stack < intList > s;
queue < intList > q;
cout << "Enter some numbers for the stack & queue:\n";
// set up an input stream iterator for cin
istream_iterator < int, ptrdiff_t > cinIter(cin), eos;
// put items onto stack and queue
for ( ; cinIter != eos; cinIter++) {
        s.push(*cinIter);
        q.push(*cinIter);
}
```

Stack/Queue Example (Part 2)

```
// pop items from the stack and print them out
cout << "Here's what you get when "
                  << " you pop the stack: ";
while(!s.empty()) {
        cout << s.top() << " ";
        s.pop();
}
cout << "\n\nHere's what you get when"
                  << " you pop the queue: ";
while(!q.empty()) {
        cout << q.front() << " ";
        q.pop();
}
cout << "\n\n";
```

Extended example: Counting words

- Read in a file, identify all words that appear more than once, list them with number of appearances
- Algorithm:
  - read all words into a vector of strings
  - sort the vector
  - look for repeated words, count them, put them into a map
  - index in map is # of appearances; data is list of words

word map data structure

count	word lists
4	(and, or, of)
3	(we, can)
2	(STL, template, Utica, pistachio)

Acknowledgment: example inspired by the "anagram" program of Musser and Saini.

word count (part 1)

```
// First, open input file, set up iterator for reading
cout << "Enter name of file to process: ";
string inFileName;
cin >> inFileName;
ifstream inFile(inFileName.c_str());
istream_iterator < string, ptrdiff_t >
      inFileIter(inFile), eos;
// Set up wordVector to hold the string input, and get
// it from input file,
// then sort it to prepare for processing
typedef vector < string > stringVector;
stringVector wordVector;
copy(inFileIter, eos, back_inserter(wordVector));
cout << "Read in " << wordVector.size() << " words.\n";
sort(wordVector.begin(), wordVector.end());
```

word count (part 2)

```
// Now set up a map to hold the duplicated words.
// The key field is the number of times the word occurs,
// and the data field is a list of all words that occurred
// that number of times.
// The map is organized from largest to smallest.
typedef map < int, list < string >, greater < int > >
        stringCountMap;
stringCountMap wordMap;

// create an iterator "current" for wordVector,
// then process the vector,
// looking for duplicated words and entering them into wordMap
stringVector::iterator current = wordVector.begin();
while (current != wordVector.end()) {
        // set current to next occurrence of a duplicated word
        current = adjacent_find(current, wordVector.end());
        if (current == wordVector.end()) break;
```

word count (part 3)

```
// find the next word that *doesn't* match the current word,
// thus setting up an open interval [current, nextWordPos)
// that marks all matching words
stringVector::iterator nextWordPos =
        find_if(current+1, wordVector.end(),
                not1(bind1st(equal_to<string>(), *current)));
// add the word to the list at the appropriate map entry,
// determined by the number of times it's in the vector
wordMap[nextWordPos-current].push_back(*current);
// continue processing the vector at the next word
current = nextWordPos;
}
```

word count (part 4)

```
// set up an output iterator and present results
ostream_iterator <  string > coutIter(cout, "\n");
cout << "map file is: \n";

// iterate through the map, print the key
// (which indicates the number of
// occurrences of the word), then copy the list
// of words to output
stringCountMap::const_iterator wordMapIter =
   wordMap.begin();
for (; wordMapIter != wordMap.end(); ++wordMapIter) {
    cout << "Count " << (*wordMapIter).first << ":\n";
    copy((*wordMapIter).second.begin(),
            (*wordMapIter).second.end(), coutIter);
    cout << "----\n\n";
}
```

Sample output

```
Enter name of file to process: cx6.cpp
Read in 441 words.
map file is:
Count 26:
//
----
Count 23:
the
----
Count 12:
of
----
Count 9:
<<
----
Count 8:
#include
to
----
Count 7:
<
and
string
----
Count 6:
in
words
----
etc.
```

Extending the STL

- Not standardized but available
  - » hash_set
  - » hash_map
  - » hash_multiset
  - » hash_multimap
- Like set... but have a (self reorgainzing) hashed implementation
- Constant average time for insert/erase

STL in Java

- ObjectSpace has developed an equivalent library for Java
- (JGL) Java Generic Library
- Public domain, available on internet.
- Depends on run-time typing instead of compile time typing, but is otherwise equivalent.

Resources

- http://csis.pace.edu/~bergin
- http://www.objectspace.com
- http://www.cs.rpi.edu/~musser/stl.html
- http://weber.u.washington.edu/~bytewave/bytewave_stl.html
- ftp.cs.rpi.edu/pub/stl
- http://www.sgi.com/Technology/STL/
- http://www.cs.brown.edu/people/jak/programming/stl-tutorial/home.html

Books

- Data Structures Programming with the STL, Bergin, Springer-Verlag (to appear)
- STL Tutorial and Reference Guide, Musser and Saini, Addison-Wesley, 1996
- The STL <primer>, Glass and Schuchert, Prentice-Hall, 1996
- The Standard Template Library, Plauger, Stepanov, and Musser, Prentice-Hall, 1996

map and multimap

- Ordered set (multiset) of key-value pairs
- Kept in key order
- O(lg n) inserts and deletions
- Bidirectional iterators
- Good choice for dictionaries, property lists, & finite functions as long as keys have comparison operation

vector

- Expandable array -- operator[]
- push_back, pop_back
- Average O(1) insert at end.
- O(n) insert in middle
- Random Access Iterators
- Fastest (average) container for most purposes.

deque

- Expandable “array” at both ends
- push_front, pop_front
- Average O(1) insert at both ends
- Linear insert in middle
- Random Access Iterators
- Good choice for queues & such.

list

- Doubly linked list
- O(1) inserts everywhere, but slower on average than vector and deque
- Bidirectional iterators
- Some specialized algorithms (sort).

set and multiset

- Sorted set (multiset) of values
- O(lg n) inserts and deletions
  - Balanced binary search tree
- Sorted with respect to operator< or any user defined comparison operator
- Bidirectional iterators
- Good choice if elements must stay in order.

All Iterators Provide

- operator*
  - may be readonly or read/write
- copy constructor
- operator++ and operator++(int)
- operator== and operator!=
- Most provide operator=

Specialized Iterators

- Forward
  - provide operator=
- Bidirectional (extend forward)
  - provide operator-- and operator--(int)
- Random Access (extend bidirectional)
  - provide operator<..., operator+=..., operator-