Porter Scobey

http://cs.stmarys.ca/~porter/csc/ref/stl/index_algorithms.html

Stanford 106L, Standard C++ Programming Laboratory

http://web.stanford.edu/class/cs106l/

topcoder's tutorial, Power up C++ with the STL, part I and II

https://www.topcoder.com/community/data-science/data-science-tutorials/power-up-c-with-the-standard-template-library-part-1/

STL Algorithms

<algorithm>, <numeric>, <iterator>, <functional>
<cctype>, <cmath>

C++, a multi-paradigm programming language, besides being procedural and object-oriented, is very much ***functional*** with STL

Pei-yih Ting
NTOU CS

Contents

Abstract away some chores

- Commonly seen procedural piece of codes

low-level mechanical steps

High level abstract thoughts

```
#include <iostream>
#include <fstream>
#include <set>
using namespace std;
int main() {
   ifstream input("data.txt");
   multiset<int> values;

   int currValue;
   while (input >> currValue)
      values.insert(currValue);

   double total = 0.;
   for (multiset<int>::iterator itr = values.begin(); itr != values.end(); ++itr)
      total += *itr;

   cout << "Average = " << total / values.size() << endl;
   return 0;
}
```

Functional abstraction

❶ **Read the contents of the file**

```
copy(istream_iterator<int>(input), istream_iterator<int>(),
   inserter(values, values.begin()));
```

map <algorithm>

❷ **Add the values together**

```
double total = accumulate(values.begin(), values.end(), 0.0);
```

reduce <numeric>

❸ **Calculate the average**

data.txt

```
100
95
92
89
100
...
```

Average = …

Functional Language

- Mathematically a **functional** is a function of a function, or higher order function, ex. Integration, Derivative, arc length, …

- Functional programming is a **declarative** programming paradigm which models **computations** as **the evaluation of mathematical functions** and avoid **changing state /** mutable data, i.e. programming is done with <u>expressions</u> or <u>declarations</u> instead of statements. The output value of a function depends only on the arguments that are input to the function without side effects such that it is easier to understand and predict the behavior of a program.

- **Functional and Object-oriented styles are not easy to combine.**
- Bjarne Stroustrup's: C++ was designed to allow programmers to **switch** between paradigms as needed. The language is not designed to make it easy for combining different paradigms. Most of Stroustrup's examples regarding OOP touch the STL very little. He creates very distinct layers.

Tools for Functional Abstraction

- **Algorithms:** optimized machinery
  - Map:
    transform / copy / for_each / replace / sort / partition
  - Filter:
    removal (find and erase)
  - Reduce:
    accumulate / min_element / count / equal / search / selection

customized with callable objects (functions and functors)

- **Core data structure:** container

accumulate()

Your first high-level machinery

- #include <numeric>
- accumulate sums up the elements in a **range** and returns the result

```
multiset<int> values;          [begin(), end())      initial value
…
double total = accumulate(values.begin(), values.end(), 0.0);
accumulate(values.lower_bound(42), values.upperbound(99), 0.0);
```

Your first higher order function (user customized)

- accumulate is a general-purpose function for **transforming a collection of elements into a single value** (in functional language terms: reduce / collect / convert / join / fold)

```
int findLess(int smallestSoFar, int current) {
   return current < smallestSoFar ? current : smallestSoFar;   }
int smallest = accumulate(values.begin(), values.end(),
       <limits>        numeric_limits<int>::max(), findLess);
```


Advantages

- **Simplicity**:
  - Leverage off of code that is already written for you rather than reinventing the code from scratch; don't duplicate code
- **Correctness**:
  - already tested without manual mistakes
- **Speed**:
  - STL algorithms are optimized such that they are faster than most code you could write by hand
- **Clarity**:
  - With a customized for loop, you would have to read each line in the loop before you understood what the code did.

Algorithm Naming Conventions

- More than 50 STL algorithms (<algorithm> and <numeric>)
- **xxx_if** (replace_if, count_if, …): means the algorithm will perform a task on elements only if they meet a certain criterion
  require a predicate function: accepts an element and returns a bool
  e.g. bool **isEven**(int value) { return value %2 == 0; }
  cout << **count_if**(myVec.begin(), myVec.end(), **isEven**);
- **xxx_copy** (remove_copy, partial_sort_copy, …): performs some task on a range of data and store the result in another location (immutable)
  e.g. int iarray[] = {0, 1, 2, 3, 3, 4, 3}; vector<int> myV(7);
  reverse_copy(iarray, iarray+7, myV.begin());
- **xxx_n** (generate_n, search_n, …): performs a certain operation n times (on n elements in the container)
  e.g. **fill_n**(myDeque.begin(), 10, 0);
  vector<int>::iterator it=**search_n**(myV.begin(), myV.end(), 2, 3);   Two consecutive 3

Iterator Categories

- STL iterators are categorized based on their relative power
- Functionalities: minimal (I/O) => maximal (Random-Access)
  - For example, iterators for vector/deque support container.begin()**+n**, while iterators for set/map only support **++** (efficiency reasons)
- Categories:
  - **Output** Iterators: *itr = value, referred object is write-only, ++, no --, +=, -
  - **Input** Iterators: value = *itr, referred object is read-only, ++, no --, +=, -
  - **Forward** Iterators: both *itr = value and value = *itr, ++ is OK but not --
  - **Bidirectional** Iterators: iterators of **map/set/list**, ++, -- are OK but not +=, -
  - **Random-Access** Iterators: iterators of **vector/deque**, ++, --, +=, -, <, >, +, []
- If an algorithm requires a Forward Iterator, you can provided it with a Forward/Bidirectional/Random-Access iterator.
- If an algorithm demands an Input iterator, it guarantees that the container pointed by the Input iterator is read-only by this algorithm.

Iterator Categories (cont'd)

Random-Access Iterators

itr + distance;
itr += distance; std::advance(itr, distance);
itr1 < itr2; std::distance(itr1, itr2); itr2 - itr1;
itr[myIndex]; *(itr + myIndex);

Bidirectional Iterators

--itr;

Forward Iterators

Input Iterators

val = *itr;
++itr;

Output Iterators

*itr = val;
++itr;

Reordering Algorithms

- **sort** // random-access iterators
  - **sort(myVector.begin(), myVector.end());**
    // i.e. vector or deque only, cannot sort list, set or map
    // each element must provide **operator<** or **comparison function**
  - ex.bool compStrLen(const string &one, const string &two) { // or pass by value
    return one.length() < two.length();
    } use pair to do multifield comparison
    sort(myVector.begin(), myVector.end(), compStrLen);
- **stable_sort, partial_sort, partial_sort_copy, is_sorted, nth_element**
- **partition** // bidirectional iterators
  - bool isOdd(int i) { return (i%2)==1; }

```
vector<int> myvector;
for (int i=1; i<10; ++i) myvector.push_back(i); // 1 2 3 4 5 6 7 8 9
vector<int>::iterator bound =
std::partition(myvector.begin(), myvector.end(), isOdd);
// possible result: 1 9 3 7 5 6 4 8 2
```

bound ------> (points to 6)

- **stable_partition, is_partitioned**

Reordering Algorithms (cont'd)

- **reverse** // bidirectional iterators
  - **reverse(myVector.begin(), myVector.end());**
- **random_shuffle** // random-access iterators
  - **random_shuffle(myVector.begin(), myVector.end());**
- **shuffle** (C++11) // random-access iterators
  - unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
    shuffle(myVec.begin(), myVec.end(), std::default_random_engine(seed));
- **rotate** // forward iterators
  - **rotate(v.begin(), v.begin()+2, v.end());** // begin, middle, end
    // (0, 1, **2**, 3, 4, 5) => (2, 3, 4, 5, 0, 1)
- **next_permutation** // bidirectional iterators, operator<
  - int v[] = {1, 4, 2};
    next_permutation(v, v+3); // (1, 4, 2) => (2, 1, 4)
    find next with carry
- **prev_permutation**

Other Utilities

- ✧ **min(a,b)** ✧ **max(a,b)**
  - return the smaller one of a and b
  - cout << **min(2,1)** << ' ' << **min(3.5, 2.1)** << ' ' << **min('d', 'b')**; //1 2.1 b
- ✧ **min_element** // forward iterators, operator<
  - return the iterator of the smallest element in range [first, end)
  - bool **myfn**(int i, int j) { return i<j; } ✧ **max_element**
    struct { bool operator() (int i,int j) { return i<j; } } **myobj**;
    int myints[] = {3,7,2,5,6,4,9};
    cout << ***min_element(myints,myints+7)**; // 2, operator<
    cout << ***min_element(myints,myints+7,myfn)**; // 2
    cout << ***min_element(myints,myints+7,myobj)**; // 2
- ✧ **merge** // **sorted** range, input iterators, operator<
  - int a[] = {10,5,15,25,20}; int b[] = {50,40,30,20,10}; vector<int> c(10);
    sort(a, a+5); sort(b, b+5);
    merge(a, a+5, b, b+5, c.begin());
- ✧ **inplace_merge(first, middle, last)** // bidirectional iterators // [first, middle) + [middle, last) // **sorted** ranges, operator<

Other Utilities (cont'd)

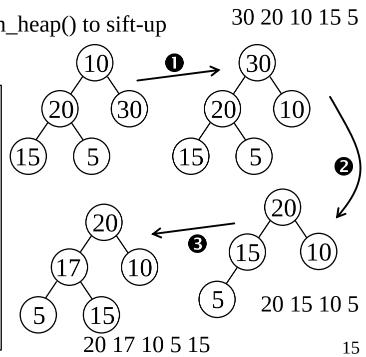
- ✧ **set_union** // union of 2 **sorted** ranges, input iterators, operator<
  - return an output iterator that is the end of the constructed sorted ranges
  - int a[] = {10,5,15,25,20}; int b[] = {50,40,30,20,10}; vector<int> c(10);
    sort(a, a+5); sort(b, b+5); // 5 10 15 20 25 30 40 50 0 0
    vector<int>::iterator endIter = **set_union(a, a+5, b, b+5, c.begin());**
    cout << *(endIter-1) << endl; // 50
    c.resize(endIter-c.begin()); // 5 10 15 20 25 30 40 50
- ✧ **set_intersection** // intersection of 2 **sorted** ranges, input iterators, operator<
  - vector<int>::iterator endIter = **set_intersection(a, a+5, b, b+5, c.begin());**
    cout << *(endIter-1) << endl; // 20 // 10 20 0 0 0 0 0 0 0 0
    c.resize(endIter-c.begin()); // 10 20
- ✧ **bitset**
  - bitset<5> foo(string("01011"));
    foo[0] = 0; /* LSB 01010 */ foo[1] = foo[0]; /* 01000 */
    foo.flip(1); /* 01010 */ foo.flip(); /* 10101 */
    cout << foo << ' ' << boolalpha << foo.test(3) << ' ' << foo.count() // 10101 false 3

Max Heap

#include <queue>
std::priority_queue<int>
push()
empty(), top(), pop()

- ✧ Maintain a max heap in a vector or deque
  - creation (heapify): **make_heap** // random-access iterators, **operator<**
  - extract maximum: **pop_heap**
    - does not remove maximum element from the container
    - move the maximum to the end of the range, use v.pop_back() to remove it
    - does not return anything, use v.front() beforehand or use v.back() afterward
  - insert element: **push_heap**
    1. v.push_back() to append the element, 2. push_heap() to sift-up
  - sort the heap: **sort_heap**

```
int myints[] = {10,20,30,15,5};
vector<int> v(myints,myints+5);
❶ make_heap(v.begin(),v.end());
cout << v.front() << endl; // 30
❷ pop_heap(v.begin(),v.end()); v.pop_back();
❸ v.push_back(17); push_heap(v.begin(),v.end())
sort_heap(v.begin(),v.end()); // no longer a heap
```



Searching Algorithms

- ✧ **InputItr find(InputItr first, InputItr last, const Type& value)**
  - search for **value** in designated range [**first**, **last**)
  - return an iterator to the first element found; use a while loop to find all
  - returns **last** iterator if nothing found
  - if (**find(myVec.begin(), myVec.end(), 137)** != myVec.end()) …
  - avoid using find() on set or map, use set::find() or map::find() for efficiency

---

- ✧ **iterator_traits<InputItr::difference_type> count(InputItr first, InputItr last, const Type& value)**
  - ✧ return the number of elements **x == value** in the designated range [**first**, **last**)

- ✧ **bool binary_search(RandItr first, RandItr last, const Type& value)**
  - search for **value** in the designated **sorted** range, [**first**, **last**), delineating by two random-access iterators, i.e. iterators of vector or deque (map or set are sorted, their find() are efficient, i.e. log n)
  - return true if found; false otherwise
  - if (**binary_search(myVec.begin(), myVec.end(), 137)**) … // found

Searching Algorithms (cont'd)

- **ForwardItr lower_bound(ForwardItr first, ForwardItr last, const Type& value)**
  - Find the first element **x ≥ value** in the designated **sorted** range [**first**, **last**)
  - itr = **lower_bound(myVec.begin(), myVec.end(), 137);**
  - return an iterator to the first element satisfying **x ≥ value**
  - if (itr == last) … // all elements in [first, last) satisfy x < value
    else if (*itr == 137) … // 137 is found
    else … // *itr > 137

- **ForwardItr upper_bound(ForwardItr first, ForwardItr last, const Type& value)**
  - Find the first element **x > value** in the designated **sorted** range [**first**, **last**)
  - itr = **upper_bound(myVec.begin(), myVec.end(), 137);**
  - return an iterator to the first element satisfying **x > value**

both algorithms are O(log n)

Searching Algorithms (cont'd)

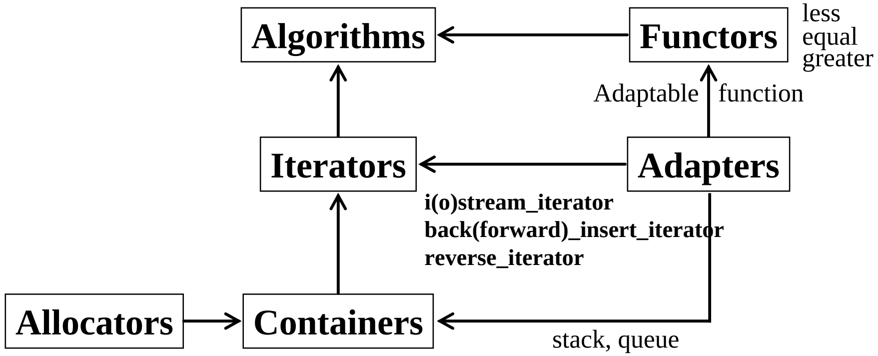
- **ForwardItr search(ForwardItr first1, ForwardItr last1, ForwardItr first2, ForwardItr last2)**
  - Searches the range [**first1**, **last1**) for the first occurrence of the subsequence defined by [**first2**, **last2**), **operator==** is required
  - returns an iterator to its first element in [first1, last1), or last1 if no occurrence is found.
  - e.g. [first1, last1) = (10, 20, 30, **40**, **50**, **60**, **70**, 80)
    [first2, last2) = (**40**, **50**, **60**, **70**)

  Invoking search(first1, last1, first2, last2) returns first1+3

- **ForwardItr search_n(ForwardItr first, ForwardItr last, Size n, const Type& value)**
  - find first subsequence of n **value**'s

- **bool includes(InItr first1, InItr last1, InItr first2, InItr last2)**
  - Two **sorted** ranges [**first1**, **last1**), [**first2**, **last2**)
  - Returns whether every elements in [**first2**, **last2**) is also in [**first1**, **last1**)
  - e.g. (1,**2**,3,**4**,5) includes (**2,4**)

Six Parts of STL



- **Containers** rely on the **allocators** for memory and support **iterators**
- **Iterators** can be used intimately in conjunction with the **algorithms**.
- **Functors** provide special extensions for the **algorithms**.
- **Adapters** can produce modified **functors**, **iterators**, and **containers**.

Iterator Adaptors - **<iterator>**

Iterator adaptor does not actually point to elements in a container. It helps inserting/extracting elements from a container or stream.

- **ostream_iterator<type>** / **ostreambuf_iterator<char>**: formatted / unformatted output iterator, attached to an ostream, use dereference operator to write data to the output stream, useful to STL algorithms

```
ostream_iterator<int> myItr(cout, " "); *myItr = 123; myItr++;
vector<int> myVec;
copy(myVec.begin(), myVec.end(), ostream_iterator<int>(cout, "\n"));
```

- **istream_iterator<type>** / **istreambuf_iterator<char>**: formatted / unformatted input iterator, attached to an istream, use dereference operator to read data from the input stream, useful to STL algorithms

```
copy(istreambuf_iterator<char>(cin), istreambuf_iterator<char>(),
     ostreambuf_iterator<char>(cout) ); // one line stream copy
istream_iterator<int> itr(cin), endItr;
int x; do x = *itr++, cout << x; while (itr!=endItr);
```

1 2 3 4^Z
1 2 3 4

Note: **endItr** marks the end, is not attached to any input stream

Iterator Adaptors - **<iterator>**

- Many STL algorithms take in ranges of data and **produce new data ranges** as output. The results are overwritten to the destination. You must ensure that the destination has enough space to hold the results.
- **back_insert_iterator<Container>**, (vector, deque, list) **front_insert_iterator<Container>**, (deque, list) and **insert_iterator<Container>** are **output** iterator adaptors simulated with container's push_back(), push_front(), and insert() members
-

```
vector<int> myVector; // no need to allocate space beforehand
back_insert_iterator<vector<int> > itr(myVector);
for (int i=0; i<10; ++i) *itr++ = i;
int x[] = {10, 11, 12};                    0,1,2,3,4,5,6,7,8,9,12,11,10
reverse_copy(x, x+3, itr);
// reverse_copy(x, x+3, back_insert_iterator<vector<int> >(myVector));
// reverse_copy(x, x+3, back_inserter(myVector)); // a template function
copy(myVector.begin(), myVector.end(), ostream_iterator<int>(cout,","));
```


Iterator Adaptors - **<iterator>**

- set does not support front_insert_iterator or back_insert_iterator, only supports insert_iterator<Container> iter(container, iterator)

```
set<int> result;
set_union(set1.begin(), set1.end(), set2.begin(), set2.end(),
          inserter(result, result.begin()));
// the 2nd argument is an iterator pointing to the insertion point
// does not make sense for set or map, but is meaningful for vector, deque, or list
```

- Summary

	back_insert_iterator<vector<int> > itr(myVector); back_insert_iterator<deque<char> > itr = back_inserter(myDeque);
	front_insert_iterator<deque<int> > itr(myIntDeque); front_insert_iterator<deque<char> > itr = front_inserter(myDeque);
	insert_iterator<set<int> > itr(mySet, mySet.begin()); insert_iterator<set<int> > itr = inserter(mySet, mySet.begin());
formatted	ostream_iterator<int> itr(cout, " "); ostream_iterator<char> itr(cout); ostream_iterator<double> itr(myStream, "\n");
formatted	istream_iterator<int> itr(cin); istream_iterator<int> endItr; // Special end of stream value
unformatted	ostreambuf_iterator<char> itr(cout); // Write to cout

unformatted	istreambuf_iterator<char> itr(cin); // Read data from cin
	istreambuf_iterator<char> endItr; // Special end of stream value

Removal Algorithms

- Removal algorithms **do not remove** elements from containers; they only **shuffle down** all elements that need to be erased.
  - They accept range specified by *iterators*, not *containers*, and thus do not know how to erase elements from containers.
  - They return *iterators* to the first element that needs to be erased.
- ex1

```
int x[] = {218, 137, 130, 149, 137, 255};          218 130 149 255 137 255
vector<int> myvec;
copy(x, x+6, back_inserter(myvec));
myvec.erase(remove(myvec.begin(), myvec.end(), 137), myvec.end());
copy(myvec.begin(), myvec.end(), output_iterator<int>(cout, " "));
```

Output: **218 130 149 255**   Note: myvec.erase(myvec.end()) causes runtime error
myvec.erase(myvec.end(), myvec.end()) is fine (empty range)

- ex2

```
string stripPunctuation(string input) {
    input.erase(remove_if(input.begin(), input.end(), ::ispunct), input.end());
    return input;
}                                             <cctype>
```


Removal Algorithms (cont'd)

- **remove_copy, remove_copy_if**
  - copy the elements that aren't removed into another container, operator==

```
int myints[] = {10,20,30,30,20,10,10,20};   // 10 20 30 30 20 10 10 20
vector<int> myvector(8);                    // must ensure large enough
vector<int>::iterator iter =                              iter
remove_copy(myints, myints+8,
            myvector.begin(),               // 10 30 30 10 10 0 0 0
            20);
myvector.erase(iter, myvector.end());       //  10 30 30 10 10
```

- **unique_copy**
  - returns an iterator pointing to the end of the copied range, which contains no consecutive duplicates.

```
int myints[] = {10,20,20,20,30,30,20,20,10};
vector<int> myvector(9);                                 // 0  0  0  0  0  0  0  0  0
vector<int>::iterator iter =                                        iter
unique_copy(myints, myints+9, myvector.begin());  // 10 20 30 20 10 0  0  0  0
```

Functional Thinking

- Note: You are not going to master STL <algorithm>, <numeric>, <iterator>, or <functional> through your *procedural* or *object-oriented* intuitions!!!

Can you figure out the way to use a chainsaw in place of the ax if what you ever seen is an ax in working!!!!

Paradigm shift!!!

Functional Thinking (cont'd)

- *Functional Thinking: Paradigm over Syntax*, Neal Ford, 2014
- *Becoming Functional: Steps for transforming to a functional programmer*, Joshua Backfield, 2014
  - Cede control over low-level details to the language/runtime (e.g., use automatic garbage collection, automatic parallelism, iteration of containers, control of iterations)
  - Prefer higher-level abstractions (highly optimized machineries) customized with callable objects together operated upon key data structures (generic containers),
  - Common building blocks: filter, fold/reduce/search/selection, map/sort/partition, closures (lambda expressions)
  - Rooted on Lambda calculus. Prefer immutables and construct programs with expressions like real mathematical functions. Avoid mutable state (the moving parts) or subroutine-like processes.
  - Stop thinking of low-level details of implementation and start focusing on the problem domain and on the results across steps (gradual transformation of input data toward the results)

Functional Thinking (cont'd)

- Michael Feather
  OO makes codes understandable by encapsulating moving parts,
  FP makes codes understandable by removing moving parts.
- General characteristics of functional programming
  - Avoid mutable state, stateless transformation of immutable things
  - Recursion
  - Higher-order functions, partial functions, currying
  - Function composition
  - Lazy evaluation
  - Pattern matching
- Functional Languages: Common LISP, ML, Scheme, Erlang, Haskell, F#
- Java-based: Scala, Clojure, Java8, Groovy, Functional Java
- Languages adopting FP paradigms: Perl, PHP, Ruby on Rail, JavaScript, Python, R, C#, and C++

Again, why going FP?

- **Cloud computing**: **Google Map/Reduce**
- **Big Data**: Statistics Computing Language – **R**
- **Cool !!!**   - **Fashion !!!**

- Behind all these:
  the real motivation is **parallelism**:
  - Multicore Computing
  - GPU and Heterogeneous Computing
  - Cloud and Distributed computing
- Inherent **immutability** of Functional programming is a very good starting point for exploiting H/W parallelism.

Map / Filter / Reduce

STL — **Functional Language**

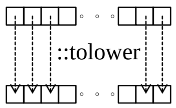
- transform() --- **map** in Scala or Closure; collect in Groovy
  copy()
  for_each()
  replace()
  sort()
  partition()
- remove() + erase() --- **filter** in Scala or Closure
- accumulate() --- **reduce** in Scala or Closure; convert; collect in Java8; inject, join in Groovy; fold in Functional Java
  count()
  equal()
  search()
  selection()
  min_element()

Mapping Algorithms

- **transform**: applies a unary function to a range of elements and stores the result in the destination range (or 2 ranges for a binary function)

```
string  convertToLowerCase(string text) {
    transform(text.begin(), text.end(), text.begin(), ::tolower);
    return text;
}
```

inplace




```
int toInt(int ch) { return ch>='a' ? ch - 'a' : ch - 'A' ; }
string  convertToInteger(string text, vector<int> &dest) {
    transform(text.begin(), text.end(), back_inserter(dest), toInt);
    return text;
}
```

the result could be another type

- **for_each**: applies a function to a range of elements

```
void toLower(int &ch) { ch = ch<='Z' ? ch - 'A' + 'a' : ch; }
string  convertToLowerCase(string text) {
    for_each(text.begin(), text.end(), toLower);
    return text;
}
```

call-by-value parameter

for_each returns the copied function object

Mapping Algorithms (cont'd)

- **replace**(ForwardItr start, ForwardItr end, const Type & toReplace, const Type& replaceWith)

```
int myints[] = {10, 20, 30, 30, 20, 10, 10, 20};
vector<int> myvector(myints, myints+8);              // 10 20 30 30 20 10 10 20
replace(myvector.begin(), myvector.end(), 20, 99); // 10 99 30 30 99 10 10 99
```

- **replace_if**(ForwardItr start, ForwardItr end, Predicate fn, const Type& replaceWith)

```
bool isOdd (int i) { return ((i%2)==1); }
int myints[] = {10, 11, 30, 30, 13, 10};
vector<int> myvector(myints, myints+8);                    // 10 11 30 30 13 10
replace_if(myvector.begin(), myvector.end(), isOdd, 0); // 10  0 30 30  0 10
```

- **generate**(ForwardItr start, ForwardItr end, Generator fn)

```
int randomNumber() { return (std::rand()%100); }
srand(unsigned(std::time(0)));   vector<int> myvector(8);
generate(myvector.begin(), myvector.end(), randomNumber);
```

- **generate_n**(OutputItr start, size_t n, Generator fn)

for_each

```
#include <iostream>
#include <algorithm>
using namespace std;
template <class T>
class print {
public:
    print(ostream &os)
        : m_os(os), m_count(0) {}
    void operator()(const T &t) {
        m_os << t << ' ';
        ++m_count;
    }
    int count() { return m_count; }
private:
    ostream &m_os;
    int m_count;
};
```

```
int main()
{
    int array[] = {1, 4, 2, 8, 5, 7};
    const int n = sizeof(array) / sizeof(int);
    print<int> f =
        for_each(array, array+n, print<int>(cout));
    cout << endl << f.count()
        << " objects printed." << endl;
    return 0;
}
```

STL Abstractions

- **Containers** abstract away the differences of underlying basic types: the same vector implementation is used for **int**s or **string**s. An algorithm that handles elements in a vector does not concern the actual type stored in that vector.
- **Iterators** abstract away which container was used. For example, the same random-access iterator implementation is used for vector or deque and provides uniform interfaces to various algorithms.
- **Algorithms** (customized with the **plugged-in function objects**) are abstract mechanisms that focus on solving the general structure of a problem instead of the particular container or the specific data type in the container.

Things to Remember

when using STL algorithms

- *Prefer a **member function** to a similarly named algorithm for performing a given task*
  e.g. std::set::lower_bound() vs. std::lower_bound()
- *Don't be afraid to use **ordinary array pointers** in a manner analogous to the use of iterators, where appropriate*
  e.g. int data[] = {5,4,2,3,1}; sort(data, data+5);
- *You can generally use a **more powerful iterator** in place of a less powerful one, if it is more convenient or "natural" to do so.*
  e.g. replace_n() requires output iterator, the ostream_iterator suffices, but the bidirectional iterator of list/set/map is also good, let along the random-access iterator of vector/deque
- *Ensure a **container's size is large enough** to accept all components being transferred into it.*
  e.g. vector<int> v; back_insert_iterator<vector<int> > iter(v);

Palindrome

- A palindrome is a word or phrase that is the same when read forwards or backwards, such as "racecar" or "Malayalam.", ignoring spaces, punctuation, and capitalization.
- Procedural way

```
bool IsPalindrome(string input) {
   for (int k=0; k<input.size()/2; ++k)
      if (input[k]!=input[input.length()-1-k])
         return false;
   return true;
}
```

- 1st STL version

```
bool IsPalindrome(string input) {
   string reversed = input;
   reverse(reversed.begin(), reversed.end());
   return reversed == input;
}
```

```
string reversed;
reverse_copy(input.begin(),
             input.end(),
             reversed);
return reversed == input;
```

plain, narrative, but less efficient

Palindrome (cont'd)

- 2nd STL version: use **reverse_iterator** and **equal**

```
bool IsPalindrome(string input) {
   return equal(input.begin(), input.begin()+input.size()/2, input.rbegin());
}
```

- More: stripping out everything except alphabetic char

```
#include <cctype> // isalpha()
#include <algorithm> // remove_if(), equal()
bool  IsNotAlpha(char ch) {
   return !isalpha(ch);
}
bool IsPalindrome(string input) {
   input.erase(remove_if(input.begin(), input.end(), IsNotAlpha), input.end());
   return equal(input.begin(), input.begin()+input.size()/2, input.rbegin());
}
```

Word Palindrome

- Basic steps
  1. Strip out everything except letters and spaces, convert to uppercase
  2. Break up the input into a list of words
  3. Return whether the list is the same forwards and backwards

```
bool  IsNotAlphaOrSpace(char ch) { return !isalpha(ch) && !isspace(ch); }
bool IsWordPalindrome(string input)
{
   input.erase(remove_if(input.begin(), input.end(), IsNotAlphaOrSpace),
                  input.end());
   transform(input.begin(), input.end(), input.begin(), ::toupper);
   stringstream tokenizer(input);
   vector<string> tokens;
   tokens.insert(tokens.begin(),  istream_iterator<string>(tokenizer),
                                          istream_iterator<string>());
   return equal(tokens.begin(), tokens.begin()+tokens.size()/2, tokens.rbegin());
}
```

Clang or GNU g++ has tolower()/toupper() in <locale> header also

<cctype>

int isalnum(int c)	Check if character is alphanumeric
int isalpha(int c)	Check if character is alphabetic
int isblank(int c)	Check if character is blank (C++11)
int iscntrl(int c)	Check if character is a control character
int isdigit(int c)	Check if character is decimal digit
int isgraph(int c)	Check if character has graphical representation
int islower(int c)	Check if character is lowercase letter
int isprint(int c)	Check if character is printable
int ispunct(int c)	Check if character is a punctuation character
int isspace(int c)	Check if character is a white-space
int isupper(int c)	Check if character is uppercase letter
int isxdigit(int c)	Check if character is hexadecimal digit
int isalnum(int c)	Check if character is alphanumeric
int isalpha(int c)	Check if character is alphabetic
int tolower(int c)	Convert uppercase letter to lowercase
int toupper(int c)	Convert lowercase letter to uppercase

In C++, a locale-specific **template** version of each function exists in header <locale>
use ::isalnum() to specify isalnum() in cctype

<cmath>

- Trigonometric functions

double cos(double)	Compute cosine
double sin(double)	Compute sine
double tan(double)	Compute tangent
double acos(double)	Compute arc cosine
double asin(double)	Compute arc sine
double atan(double)	Compute arc tangent
double atan2(double)	Compute arc tangent with two parameters

- Hyperbolic functions

double cosh(double)	Compute hyperbolic cosine
double sinh(double)	Compute hyperbolic sine
double tanh(double)	Compute hyperbolic tangent
double acosh(double)	Compute arc hyperbolic cosine (C++11)
double asinh(double)	Compute arc hyperbolic sine (C++11)
double atanh(double)	Compute arc hyperbolic tangent (C++11)

- Exponential and logarithmic functions

double exp(double x)	Compute exponential function, e^x
double frexp(double x, int* exp)	Get significand and exponent, x=sign*2^exp
double ldexp(double x, int exp)	x*2^exp
double log(double x)	Compute natural logarithm, w.r.t. Euler number e
double log10(double x)	Compute common logarithm
double modf(double x, double* intpart)	Break into fractional and integral parts
double exp2(double x), exp2l(x)	Compute 2^x (C++11)
double expm1(double x), expm1l(x)	Compute e^x-1 (C++11)
int ilogb(double x)	Integer binary logarithm (C++11)
int ilogb(long double x)	Returns the integral part of the logarithm of |x|, using FLT_RADIX (==2) as base
double log1p(double x)	Compute logarithm plus one, log(x+1) (C++11)
double log2(double x)	Compute binary logarithm (C++11)
double logb(double x)	Compute floating-point base logarithm, log|x| using FLT_RADIX (==2) as base (C++11)
double scalbn(double x, int n)	scalbn(x,n) = x * FLT_RADIX^n (C++11)

- Power functions

double pow(double base, double exp)	Raise to power, base^exp
double sqrt(double x)	Compute square root
double cbrt(double x)	Compute cubic root (C++11)
double hypot(double x, double y)	Compute hypotenuse (C++11)

- Error and gamma functions

double erf(double x)	Compute error function (C++11)
double erfc(double x)	Compute complementary error function (C++11)
double tgamma(double x)	Compute gamma function (C++11)
double lgamma(double x)	Compute log-gamma function (C++11)

- Rounding and remainder: ceil, floor, fmod, trunc, round, lround, llround, rint ,lrint, llrint, rearbyint, remainder, remquo
- Floating-point manipulation: copysign, nan, nextafter, nexttoward
- Minimum, maximum, difference: fdim, fmax, fmin
- Other: fabs, abs, fma
- Classification: fpclassify, isfinite, isinf, isnan, isnormal, signbit
- Comparison: isgreater, isgreaterequal, isless, islessequal, islessgreater, isunordered
- Constants: INFINITY, NAN, HUGE_VAL

Boost

- **Boost**, an excellent open source C++ libraries, provides lots of useful facilities not available in STL before C++11.
- **Boost** ==> **C++TR1** (Library extension to C++03) ==> **C++11**
  - **smart_ptrs** – manage the lifetime of referred object with reference counting (shared_ptr, shared_array, scoped_ptr, scoped_array, weak_ptr, intrusive_ptr)
  - boost::**lambda**, boost::**function**, boost::**bind** – higher order programming
  - boost::**regex** – regular expression
  - boost:: **asio** – blocking/non-blocking wait with timers, multithreading, socket
  - **FileSystem** – system independent file size, attributes, existence, directory traversal, path handling
  - template metaprogramming (boost::**mpl**)

Documents of Boost provide excellent in-depth discussions of the design decisions, constraints, and requirements that went into constructing the library.

Magic Square Solver (1/4)

- A “magic square” is a 3x3 grid in which all elements are distinct and all 3 elements in every row, column, and diagonal sum to the same number, e.g.

2	7	6
9	5	1
4	3	8

2+7+6=15
2+5+8=15
7+5+3=15
…

- A magic square can be represented as a linear vector of {1,2, …, 9}

2	7	6	9	5	1	4	3	8

```
vector<int> magicSquare(9);
for (i=0; i<9; ++i) magicSquare[i]=i+1;
```

- Use next_permutation() to generate all configurations

```
do {
   ouputConfig(magicSquare);
}
while (next_permutation(magicSquare.begin(), magicSquare.end()));
```


Magic Square Solver (2/4)

- Output a configuration

```
for (i=0; i<3; ++i)
   copy(magicSquare.begin()+3*i, magicSquare.begin()+3*i+3,
        ostream_iterator<int>(cout, " ")), cout << endl;
```

- Use **for loop** to evaluate everyone of the 8 conditions

```
const int starts[]  = {0, 3, 6, 0, 1, 2, 2, 0};
const int offsets[] = {1, 1, 1, 3, 3, 3, 2, 4};
for (i=0; i<8; ++i)
   for (sums[i]=0,j=0; j<3; ++j)
      sums[i] += magicSquare[starts[i]+j*offsets[i]];
```

- Use **count()** to validate the conjunction of all 8 conditions

```
if (8==count(sums, sums+8, 15))
   outputConfig(magicSquare);
```

Magic Square Solver (3/4)

- WHY not use accumulate() to replace the for loop?
  - Forget the low-level details, right?
  - Problem: the argument passed from accumulate() to the predicate is only value instead of the index
    - It is necessary for the predicate to have state in order to count the number of calls.
    - It is necessary to config the predicate with different ways of accumulation.

use a **functor**

```
for (i=0; i<8; ++i)
   sums[i] = accumulate(magicSquare.begin(), magicSquare.end(), 0,
                              Constraint(starts[i],offsets[i]));
```


Magic Square Solver (4/4)

```
class Constraint {
public:
   Constraint(const int start, const int offset)
      : index(0), x1(start), x2(start+offset), x3(start+2*offset) {}
   int operator()(int sum, int value) {
      if (index==x1||index==x2||index==x3)
         sum += value;
      index++;
      return sum;
   }
private:
   int index;
   const int x1, x2, x3;
};
```


Monoalphabetic Substitution Cipher

- A monoalphabetic substitution cipher is a simple form of encryption, where each of the 26 letters are mapped to another letter exclusively in the alphabet.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
K	V	D	Q	J	W	A	Y	N	E	F	C	L	R	H	U	X	I	O	G	T	Z	P	M	S	B

encryption ⇩ ⇧ decryption

For example: plaintext "THECOOKIESAREINTHEFRIDGE"
ciphertext "GYJDHHFNJOKIJNRGYJWINQAJ"

- Use **random_shuffle** to generate a map as the encoding codebook and a map as the decoding codebook

```
int encTable[26], decTable[26]; // two direct address tables (DAT)
random_shuffle(encTable, encTable+26);
for (i=0; i<26; i++) decTable[encTable[i]] = i;
```


Substitution Cipher (cont'd)

- Config **transform** with a decryption **functor**, which takes a decoding codebook and maps a ciphertext character to a plaintext character.

```
class mapping {
public:
   mapping(int table[]): DAT(table) {}
   char operator()(char &source) { return 'A'+DAT[source-'A']; }
private:
   int *DAT;
};
```

```
transform(plaintext1.begin(), plaintext1.end(),
               back_inserter(ciphertext), mapping(encTable));
```

- Config **transform** with a decryption **functor**, which takes a decoding codebook and maps a ciphertext character to a plaintext character.

```
transform(ciphertext.begin(), ciphertext.end(),
               back_inserter(plaintext2), mapping(decTable));
```

pair<T1, T2>

- **pair** is a simple, handy, and useful template of data structure used with any STL containers and algorithms.

```
template <typename T1, typename T2> struct pair {
    T1 first;  T2 second;
};
```

- Especially, it defines **operator==** and **operator<** that can be used seamlessly with containers and reordering algorithms.

```
bool operator<(pair<T1,T2>& rhs) {
    if (first<rhs.first) return true;
    else if ((first==rhs.first)&&(second<rhs.second)) return true;
    else return false; // *this >= rhs
}
```

- e.g. multi-field sorting on **pair<sring, pair<int, vector<int> > >**

```
vector<pair<string, pair<int, vector<int> > > v;
… // populate the vector v with data;
sort(v.begin(), v.end());
```


Convenient Aliases

```
typedef vector<int> vi;
typedef vector<vi> vvi;
typedef pair<int,int> ii;
typedef vector<string> vs;
typedef vector<ii> vii;
typedef vector<pair<double, ii> > vdii;
typedef vector<vii> vvii;
```

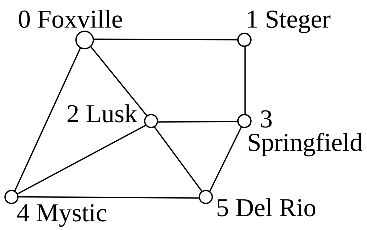
A two-dimensional 10x20 array initialized with 0
vvi matrix(10, vi(20, 0))

```
#define sz(a) int((a).size())
#define pb push_back
#define all(c) (c).begin(),(c).end()
#define tr(c,i) for(typeof((c).begin() i = (c).begin(); i != (c).end(); i++)
#define has(c,x) ((c).find(x) != (c).end())
#define hasG(c,x) (find(all(c),x) != (c).end())
```

only for GNU g++

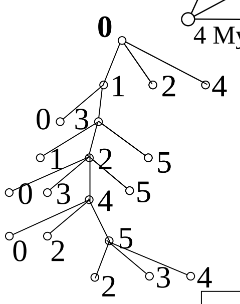
Graph Connectivity - DFS

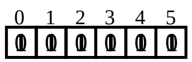
- Depth-First Search




```
vvi W;       // global graph
vi visited;
```

```
int w[][6] = {{2,4},{3},{0,4,5},
              {1},{0,2,5},{2,4}};
int wn[6] = {2,1,3,1,3,2};
int i, N = 6;
for (i=0; i<N; i++)
    W.pb(vi(w[i],w[i]+wn[i]));
```



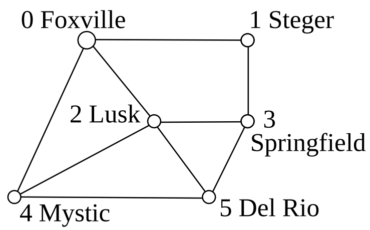


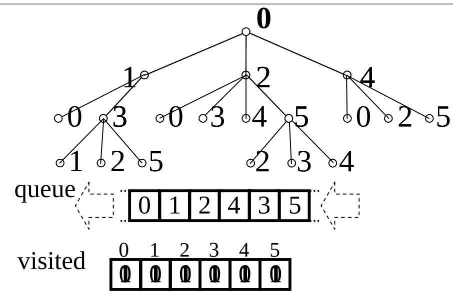

```
void dfs(int i) {
    if (!visited[i]) {
        visited[i] = 1;
        for_each(all(W[i]), dfs);
    }
}
```

```
visited = vi(N, 0); dfs(0); // start from vertex 0
if (find(all(visited), 0) == visited.end()) cout << "Connected\n";
```


Graph Connectivity - BFS

- Breadth-First Search





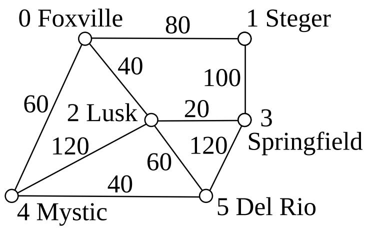

```
vi visited(N, 0);
queue<int> Q;
visited[0] = 1; Q.push(0); // start vertex is 0
while (!Q.empty()) {
    i = Q.front(); Q.pop();
    for (vi::iterator it=W[i].begin(); it!=W[i].end(); ++it)
        if (!visited[*it]) visited[*it] = 1, Q.push(*it);
}
if (find(all(visited), 0) == visited.end()) cout << "Connected\n";
```

tr (W[i], it) // only for g++

Dijkstra’s Shortest Path

```
int i, j, N = 6, w[][6][2] =
  {{{1,80},{2,40},{4,60}},
   {{0,80},{3,100}},
   {{0,40},{3,20},{4,120},{5,60}},
   {{1,100},{2,20},{5,120}},
   {{0,60},{2,120},{5,40}},
   {{2,60},{3,120},{4,40}}};
int wn[6] = {3,2,4,3,3,3};
vvii G; // weighted graph
vii tmp;

for (i=0; i<N; i++) {
  for (j=0; j<wn[i]; j++)
    tmp.push_back(pair<int,int>(w[i][j][0],w[i][j][1]));
  G.push_back(tmp);
}
```



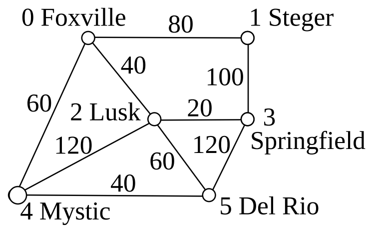
Dijkstra’s Shortest Path (cont’d)

- implementation with a **priority_queue**

```
vi D(N, 0x7fffffff);    // distance from start vertex to each vertex at the moment
priority_queue<ii,vector<ii>,greater<ii> > Q; // min heap

D[4] = 0; Q.push(ii(D[4],4)); // start vertex: 4

int v, d, v2, cost;
while (!Q.empty()) {
  ii top = Q.top(); Q.pop(); // min element
  v = top.second, d = top.first;
  if (d <= D[v]) {
    cout << v << ':' << d << endl;
    for (vii::iterator it=G[v].begin(); it!=G[v].end(); ++it) {
      v2 = it->first, cost = it->second;
      if (d + cost < D[v2])
        D[v2] = d + cost, Q.push(ii(D[v2], v2));
    }
  }
}
```



4:0 5:40 0:60 2:100 3:120 1:140

there could be multiple entries of the same vertex in the priority queue, only the one that has the least distance ever seen is considered

Dijkstra’s Shortest Path (cont’d)

- implementation with **set**

```
vi D(N, 0x7fffffff);    // distance from start vertex to each vertex at the moment

set<ii> S;  set<ii>::iterator itS;

D[4] = 0; S.insert(ii(D[4],4)); // start vertex: 4

while (!S.empty()) {
  ii top = *(S.begin()); S.erase(S.begin()); // min element
  int v = top.second, d = top.first;
  cout << v << ':' << d << endl;
  for (vii::iterator it=G[v].begin(); it!=G[v].end(); ++it) {
    int v2 = it->first, cost = it->second;
    if (D[v2] > d + cost) { // d==D[v] actually
      if ((D[v2]!=0x7fffffff)&&(itS=S.find(ii(D[v2],v2)))!=S.end()))
        S.erase(itS);
      D[v2] = d + cost; S.insert(ii(D[v2], v2));
    }
  }
}
```

4:0 5:40 0:60 2:100 3:120 1:140

might be erased earlier

guarantees no duplicated entry with the same vertex in the set

References

- ***The C++ standard library: a tutorial and reference***, Nicolai. M. Josuttis, 2nd ed., AW, 2012 (1st ed, 1999)
  - ***C++ 標準程式庫***, 侯捷, 2002 (Josuttis, 1st ed.)
- ***Generic Programming and the STL - Using and Extending the C++ Standard Template Library***, by Matthew H. Austern, AW, 1998
  - ***泛型程式設計與 STL***, 侯捷/黃俊堯合譯, 碁峰, 2001
- ***The Annotated STL Sources***,
  - ***STL 源碼剖析***, 侯捷, 碁峰, 2002
- ***Effective STL***, by Scott Meyers, AW, 2001
  - ***Effective STL 簡中***, Center of STL study
- ***Modern C++ Design Generic Programming and Design Patterns Applied***, Andrei Alexandrescu, AW, 2001
- ***Designing Components with the C++ STL***, 3rd ed., Ulrich Breymann, AW, 2002