STL Algorithms

- STL algorithms perform operations on collections of data.
  - Designed to work with containers. Can also be used with arrays.
  - They are essentially function templates.
- Some examples: find, count, search, sort, merge, for_each, transform.

find: finds the first occurrence of an element in a container

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
  int arr[] = {11, 22, 33, 33, 44, 55, 66};
  int * ptr = find(arr, arr+7, 33);
  cout << ptr-arr << endl;

  vector<int> vec = {11, 22, 33, 33, 44, 55, 66};
  auto it = find(vec.begin(), vec.end(), 33);
  cout << it-vec.begin() << endl;
}
```

Prints:

2

2

search: searches for the first occurrence of a pattern

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
  int source[] = {11, 44, 33, 11, 22, 33, 11, 22, 44};
  vector<int> pattern = {11, 22, 33};
  int * ptr = search(source, source+9, pattern.begin(), pattern.end());
  cout << ptr - source << endl;
}
```

Prints:

3

transform

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int myAbs(int n) {return (n > 0) ? n : -n;}

int main()
{
  int arr[] = {45, 2, 22, 17, 0, -30, 25, 55};
  vector<int> vec(8);
  transform(arr, arr+8, vec.begin(), myAbs);
  for (auto e : vec)
    cout << e << " ";
  cout << endl;
}
```

Prints: 45 2 22 17 0 30 25 55

transform with functors

```
class sumHelper
{
  private:
    int runningSum;
  public:
    int operator() (int n)
    {
      runningSum += n;
      return runningSum;
    }
};
```

Function Object, or a functor

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
  int arr[] = {1, 2, 3, 4, 5, 6, 7, 8};
  transform(arr, arr+8, arr, sumHelper());
  for (auto e : arr)
    cout << e << " ";
  cout << endl;
}
```

Prints: 1 3 6 10 15 21 28 36

How to define custom algorithms on STL containers?

Suppose we want to define a function template which finds whether an element is present in a container, and returns true or false

Recall:

```
template<class ElemType>
int findElem(ElemType * arr,
ElemType elem, int size)
{
    for (int i = 0; i < size; i++)
        if (arr[i] == elem)
            return true;
    return false;
}
```

```
template<class Container, class ElemType>
int findElem(Container & c, ElemType elem)
{
    for (auto & m : c)
      if (m == e)
        return true;
    return false;
}
```

This won't work, because STL containers are class templates themselves

How to define custom algorithms on STL containers?

We need to define template template parameters

```
template<template <class, class> class Container, class ElemType, class Type>
bool findElem(Container<ElemType, Type> & c, ElemType e)
{
  for (auto & m : c)
    if (m == e)
      return true;
  return false;
}
```

Prints:

1

0

1

```
int main()
{
  vector<int> v1 = {1,2,3,4,5,6};
  vector<string> v2 = {"a","b","c","d"};
  list<int> l1 = {1,2,3,4,5,6};
  cout << findElem(v1, 1) << endl;
  cout << findElem(v2, string("e")) << endl;
  cout << findElem(l1, 1) << endl;
}
```

Case study on STL Maps from Weiss, Chapter 4

- Given a collection of words *C* and a word *w* in *C,* suppose we want to find all words which are “adjacent” to *w* in *C*.
- Two words w1 and w2 are adjacent if w1 can be obtained from w2 by a single character substitution.
  - Example: dine, fine, line, wine, mine, nine, pine, vine

Quadratic time algorithm to find similar words

```
1   // Computes a map in which the keys are words and values are vectors of words
2   // that differ in only one character from the corresponding key.
3   // Uses a quadratic algorithm.
4   map<string,vector<string>> computeAdjacentWords( const vector<string> & words )
5   {
6       map<string,vector<string>> adjWords;
7
8       for( int i = 0; i < words.size( ); ++i )
9           for( int j = i + 1; j < words.size( ); ++j )
10              if( oneCharOff( words[ i ], words[ j ] ) )
11              {
12                  adjWords[ words[ i ] ].push_back( words[ j ] );
13                  adjWords[ words[ j ] ].push_back( words[ i ] );
14              }
15
16      return adjWords;
17  }
```

Weiss reports that on a dictionary containing 89,000 words, this implementation takes 97 seconds

oneCharOff()

```
1   // Returns true if word1 and word2 are the same length
2   // and differ in only one character.
3   bool oneCharOff( const string & word1, const string & word2 )
4   {
5       if( word1.length( ) != word2.length( ) )
6           return false;
7
8       int diffs = 0;
9
10      for( int i = 0; i < word1.length( ); ++i )
11          if( word1[ i ] != word2[ i ] )
12              if( ++diffs > 1 )
13                  return false;
14
15      return diffs == 1;
16  }
```

Example

dine, find, fine, line, wide, wife, wind, wine, wing

dine	fine, line, wine
find	fine, wind
fine	dine, find, line, wine
line	dine, fine, wine
wide	wine
wife	wine

wind	wine
wine	dine, fine, line, wide, wife, wind, wing
wing	wind

```
map<string,vector<string>> adjWords;
map<int,vector<string>> wordsByLength;

  // Group the words by their length
for( auto & thisWord : words )
    wordsByLength[ thisWord.length( ) ].push_back( thisWord );

  // Work on each group separately
for( auto & entry : wordsByLength )
{
    const vector<string> & groupsWords = entry.second;

    for( int i = 0; i < groupsWords.size( ); ++i )
        for( int j = i + 1; j < groupsWords.size( ); ++j )
            if( oneCharOff( groupsWords[ i ], groupsWords[ j ] ) )
            {
                adjWords[ groupsWords[ i ] ].push_back( groupsWords[ j ] );
                adjWords[ groupsWords[ j ] ].push_back( groupsWords[ i ] );
            }
}
```

A simple optimisation: group words by length, and search for similar words in each group only

This reduces the execution time to 19 seconds

More optimisation using even more maps

- The main idea is to find representative words obtained by deleting a single character.
- We then construct a “representatives” map whose keys are representative words, and values are collection of represented words.
  - We use the representatives map to construct the adjacent words map.
- Representatives map can be constructed in linear time.

Pseudocode

```
for each group g, containing words of length len
    for each position p (ranging from 0 to len-1)
    {
        Make an empty map<string,vector<string>> repsToWords
        for each word w
        {
            Obtain w's representative by removing position p
            Update repsToWords
        }
        Use cliques in repsToWords to update adjWords map
    }
```

Build representatives map

```
map<string,vector<string>> repToWord;

for( auto & str : groupsWords )
{
    string rep = str;
    rep.erase( i, 1 );
    repToWord[ rep ].push_back( str );
}
```

…and then build adjacent words map

```
for( auto & entry : repToWord )
{
    const vector<string> & clique = entry.second;
    if( clique.size( ) >= 2 )
        for( int p = 0; p < clique.size( ); ++p )
            for( int q = p + 1; q < clique.size( ); ++q )
            {
                adjWords[ clique[ p ] ].push_back( clique[ q ] );
                adjWords[ clique[ q ] ].push_back( clique[ p ] );
            }
}
```

This implementation runs in 2 seconds!