**Item 35. Implement simple case-insensitive string comparisons via**

**mismatch or lexicographical compare.**

First, we need a way to determine whether two characters are the same, except for their case.For that we are going to follow strcmp based approach.

//case-insensitively compare chars c1 and c2, returning - 1 if c1 < c2,

//0 if c1 == c2, and 1 if c1 > c2

int ciCharCompare(char c1, char c2){

int Ic1 = tolower(static\_cast<unsigned char>(c1));

int Ic2 = tolower(static\_cast<unsigned char>(c2));

if (Ic1 < Ic2) return -1;

if (Ic1 > Ic2) return 1;

return 0;

}

Given ciCharCompare, it's easy to write the first of our two case-insensitive string comparison functions, the one offering *a* strcmp-like interface. This function, ciStringCompare, returns a negative number, zero, or a positive number, depending on the relationship between the strings being compared. It's built around the mismatch algorithm, because mismatch identifies the first position in two ranges where the corresponding values are not the same.

Before we can call mismatch, we must satisfy its preconditions. We must make sure that if one string is shorter than the other, the shorter string is the first range passed.

int ciStringCompare(const string& s1, const string& s2){

if (s1.size() <= s2.size())

return ciStringCompareImpl(s1,s2); //Where s1 is either < or = s2

else return -ciStringCompareImpl(s2,s1); //Where s1 is always > s2.

}

In ciStringCompareImpl, the heavy lifting is performed by mismatch. It returns a pair of iterators indicating the locations in the ranges where corresponding characters first fail to match. Before continue lets discuss mismatch() algorithm example:

Case:1

// initializing vectors

    vector<**int**> v1 = { 1, 10, 15, 20 };

    vector<**int**> v2 = { 1, 10, 25, 30, 45 };

    // declaring pointer pair

    pair< vector<**int**>::iterator,   vector<**int**>::iterator > mispair;

    // using mismatch() to search for 1st mismatch

    mispair = mismatch(v1.begin(), v1.end(), v2.begin());

 // printing the mismatch pair 1st mismatch at 15 and 25

    cout << \*mispair.first << endl;   //15

    cout << \*mispair.second << endl; //25

Case:2

// initializing vectors

    vector<**int**> v1 = { 1, 10, 15, 20 };

    vector<**int**> v2 = { 1, 10, 15, 20, 45 };

bool isEqual(const int &x, const int &y) {

if (x == y) return true;

else return false;

}

pi = mismatch(v1.begin(), v1.end(), v2.begin(), isEqual);

//cout << \*pi.first << "\n"; //Not dereferenceable \*pi.first pointes to the end()

//cout << \*pi.second << "\n"; //Points to 45.

if (pi.first == v1.end()){

cout << "Vector v1 is totally consumed\n";

if (pi.second == v2.end()) {

cout << "vector v2 is totally consumed, So both are equal\n";

}else {

cout << "vector v1 & v2 are uneual( v1 < v2)\n";

}

}

Case:3

    vector<**int**> v1 = { 1, 10, 15, 20 ,45};

    vector<**int**> v2 = { 1, 10, 15, 20 };

pi = mismatch(v1.begin(), v1.end(), v2.begin(), isEqual);

//Abort the program v1.size() must be less then or equal to v2.size().

Now We can continue with our string comparison function:

int ciStringCompareImpl(const string& s1, const string& s2)

{

// PSCI = "pair of string::const\_iterator"

typedef pair<string::const\_iterator, string::const\_iterator> PSCI;

PSCI p = mismatch(s1.begin(), s1.end(), s2.begin(),

not2(ptr\_fun(ciCharCompare)));

//if true, either s1 and s2 are equal or s1 is shorter than s2.

if (p.first == s1.end()) {

if (p.second == s2.end()) return 0;

else return -1;

}

//the relationship of the strings is the same as that of the mismatched chars

return ciCharCompare(\*p.first, \*p.second);

}

Everything in function *ciStringCompareImpl* is self-explanatory except from below predicate:

***not2(ptr\_fun(ciCharCompare)).***

This predicate is responsible for returning true when the characters match, because mismatch will stop when the predicate returns false. We can't use ciCharCompare for this purpose, because it returns -1, 1, or 0, and it *returns 0 when the characters match,* just like strcmp.

If we passed ciCharCompare as the predicate to mismatch, C++ would convert ciCharCompare's return type to bool, and of course the bool equivalent of zero is false, which is precisely the opposite of what we want.

Similarly, when ciCharCompare returned 1 or –1, that would be interpreted as true, because, as in C, all nonzero integral values are considered true. Again, this would be the opposite of what we want.

To fix this semantic inversion, we will use the adaptor not2 and since this adaptor demands adaptable function object with certain typedef must be required ptr\_fun in front of ciCharCompare (Which convert predicate to well defined predicate class/ function object).

int ciCharCompare(char c1, char c2) {

int Ic1 = tolower(static\_cast<unsigned char>(c1));

int Ic2 = tolower(static\_cast<unsigned char>(c2));

if (Ic1 < Ic2) return -1;

if (Ic1 > Ic2) return 1;

return 0;

}

int ciStringCompareImpl(const string& s1, const string& s2){

typedef pair<string::const\_iterator, string::const\_iterator> PSCI;

PSCI p = mismatch(s1.begin(), s1.end(), s2.begin(),

not2(ptr\_fun(ciCharCompare)));

if (p.first == s1.end()) {

if (p.second == s2.end()) return 0;

else return -1;

}

return ciCharCompare(\*p.first, \*p.second);

}

struct ciStringCompare :public binary\_function<string, string, bool>{

bool operator()(const string& s1, const string &s2) {

if (s1.size() <= s2.size()) return ciStringCompareImpl(s1, s2);

else return -ciStringCompareImpl(s2, s1);

}

};

set<string, ciStringCompare> strSet;

strSet.insert("Rajeev");

strSet.insert("rajeeV");

strSet.insert("xyz");

**String comparison in logographical order:**

bool ciCharLess(char c1, char c2) {

return tolower(static\_cast<unsigned char> (c1)) <

tolower(static\_cast<unsigned char> (c2));

}

struct ciStringCompare :public binary\_function<string, string, bool> {

bool operator()(const string& s1, const string &s2) {

return lexicographical\_compare(s1.begin(), s1.end(), s2.begin(),

s2.end(), ciCharLess);

}

};

**Item 31. Know your sorting options.**

**std::partial\_sort**, which is used for sorting not the entire range, but only a sub-part of it.

vector<int> v = { 1, 3, 1, 10, 3, 3, 7, 7, -1 };

std::partial\_sort(v.begin(), v.begin() + 3, v.end());

Now elements of vector v arrange like below:



Partial\_sort has ability to take user defined comparator. If no comparator defined take by default operator “<”.

struct GreaterThen :public binary\_function<int, int, bool> {

bool operator()(int x, int y) {

return x > y;

}

};

//Arrange vector element in descending order.

std::partial\_sort(v.begin(), v.begin() + v.size(), v.end(),GreaterThen());

We can find the maximum and minimum element without sorting entire element of a vector.

std::partial\_sort(v.begin(), v.begin() + 1, v.end(),  greater<int>());

std::partial\_sort(v.begin(), v.begin() + 1, v.end()); //By default <

Element in first place of a vector gives maximum and minimum value respectively.

**std::sort() vs std::partial\_sort():**

If we use std::sort with a partial range, then only elements within that range will be considered for sorting, while all other elements outside the range will not be considered for this purpose, whereas with std::partial\_sort(), all the elements will be considered for sorting.

Another example, if we have a vector of Widgets and like to select the 20 highest-quality Widgets. In such scenario the partial\_sort easily identify the 20 best Widgets; the remainder can remain unsorted.

bool qualityCompare(const Widget& lhs, const Widget& rhs){

// return whether lhs's quality is greater than rhs's quality

Lhs>rhs //This arrange container element in descending order.

}

…

partial\_sort (widgets.begin(), // put the best 20 elements

widgets.begin() + 20, // (in order) at the front of

widgets.end(), // widgets

qualityCompare);

In arrangement quality of widget[0]>widget[1]>widget[2]…………….>widget[19]

Suppose we need top 20 quality widget but order is un-important for that we can use **nth\_element** .

**std::nth\_element():** STL algorithm which rearranges the list in such a way such that the element at the nth position is the one which should be at that position if we sort the list.

How to use nth\_element to make sure the best 20 Widgets are at the front of the

widgets vector ?

nth\_element (widgets.b

egin(), // put the best 20 elements

widgets.begin() + 20, // at the front of widgets,

widgets.end(), // but don't worry about

qualityCompare); // their order.

*Note: partial\_sort sorts the elements in positions 1-20, while nth\_element doesn't. Both algorithms, however, move the 20 highest-quality Widgets to the front of the vector*

Now the important question is, how they deal with equality?

For example, there are 12 elements with a quality rating of 1 (the best possible) and 15 elements with a quality rating of 2 (the next best). In that case, choosing the 20 best Widgets involves choosing the 12 with a rating of 1 and 8 of the 15 with a rating of 2.

How should partial\_sort and nth\_element determine which of the 15 to put in the top 20? Also, are they proving stable sorting option?

Both ***partial\_sort*** and ***nth\_element*** provide un-stable sorting option. If we need stable soring, we have an option for it: ***stable\_sort***.

STL provides one another feature called **partition**. It partitions the elements on **basis of condition** mentioned in its arguments. A **is\_partitioned(beg, end, condition)** is a function which returns boolean **true if container is partitioned** else returns false.

bool isEven(int x) {

return x % 2 == 0;

}

vector<int> vect = { 2, 1, 5, 6, 8, 7 };

bool res=is\_partitioned(vect.begin(), vect.end(), isEven); //false

// partitioning vector using partition()

partition(vect.begin(), vect.end(), isEven);

// Checking if vector is partitioned using is\_partitioned()

res=is\_partitioned(vect.begin(), vect.end(), isEven); //true

In our old example, need to move all the Widgets with a quality rating of 2 or better to the front of widgets, we define a function that identifies which Widgets make the grade.

bool hasAcceptableQuality(const Widget& w){

// return whether w has a quality rating of 2 or better;

}

vector<Widget>::iterator goodEnd = // move all widgets satisfying

partition(widgets.begin(), // hasAcceptableQuality to

widgets.end(), // the front of widgets, and

hasAcceptableQuality); // return an iterator to the first

// widget that isn't satisfactory

**Note**: We have an option for **stable\_partition**, if we need stable partitioning.

The algorithms sort, stable\_sort, partial\_sort, and nth\_element require random access

iterators, so they may be applied only to vectors, strings, deques, and arrays. Associative containers are always in sorted order so don’t need to apply sorting algorithm on them.

List has own sort method, but if we want to apply partial\_sort, or nth\_element on the list element there is no direct method. We need to follow below approach:

list<int> lst= { 2, 1, 5, 6, 8, 7 };

vector<int> tmp(lst.begin(), lst.end());

// Apply sort

// Copy vector content to list.

Let's summarize the sorting options.

* If you need to perform a full sort on a vector, string, deque, or array, we can use sort or ***stable\_sort***.
* If we have a vector, string, deque, or array and you need to put only the top n elements in order, ***partial\_sort*** is available.
* If we have a vector, string, deque, or array and need to identify the element at position n or need to identify the top n elements without putting them in order use ***nth\_element***.
* If we need to separate the elements of a standard sequence container into those that do and do not satisfy some criterion, then option is ***partition or stable\_partition.***
* If our data is in a list, we can use ***partition*** and ***stable\_partition*** directly, and can use list-sort in place of **sort** and **stable\_sort**. If we need to perform **partial\_sort** or ***nth\_element***, go with indirect approach.

Performance point of view (space & time) use below algorithms respectively:

1. partition 4. partial\_sort

2. stable\_partition 5. sort

3. nth\_element 6. stable\_sort

**Item 33. Be wary of remove-like algorithms on containers of pointers.**

class Widget{

public:

…

bool isCertified() const; // whether the Widget is certified

…

};

vector<Widget\*> v; // create a vector and fill it with

… // pointers to dynamically

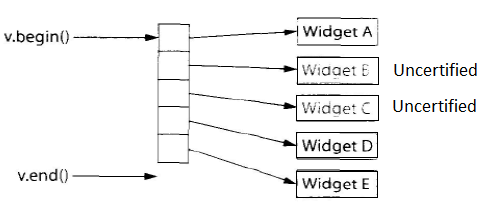
v.push\_back(new Widget); // allocated Widgets

Sometime later we don’t require uncertified Widgets. So how we can get rid form those widgets?

v.erase(remove\_if(v.begin(),v.end(), not1(ptr\_fun(Widget::isCertified)), v.end());

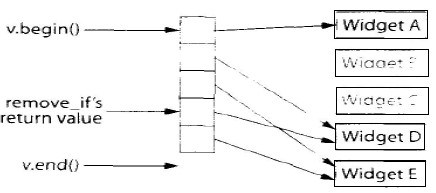
We know very well, a container filled with dynamically allocated object, then it is programmer responsibility to manually delete the object.

Let's assume that prior to the remove\_if call, v looks like this, where we have indicated the uncertified Widgets.

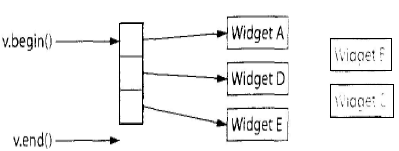


After the call to remove\_if, v will typically look like this (including the iterator

returned from remove\_if):



After remove, nothing points to the two uncertified Widgets, they can never be deleted, and their memory is leaked. Once both remove\_if and erase have returned, the situations looks as follows:



So, we need to clearly avoid using remove and similar algorithms (i.e., remove\_if and unique) on containers of dynamically allocated pointers.

One way to eliminate this problem is to delete the pointers and set them to null prior to applying the erase-remove idiom, then eliminate all the null pointers in the container:

void delAndNullifyUncertified(Widget\*& pWidget) // if \*pWidget is an

{ // uncertified Widget,

if (!pWidget->isCertified()) { // delete the pointer

delete pWidget; // and set it to null

pWidget = 0;

}

}

for\_each(v.begin(), v.end(), delAndNullifyUncertified);

Now eliminate null pointes:

v.erase( remove(v.begin(), v.end(), static\_cast<Widget\*>(0)), v.end());

**Item 34. Note which algorithms expect sorted ranges.**

A few algorithms can work with sorted or unsorted ranges, but they are most useful

when they operate on sorted ranges. Some of algorithm only required sorted range of data they are:

binary\_search lower\_bound upper\_bound equal\_range

set\_union Set\_intersection set\_difference set\_symmetric\_difference

Merge inplace\_merge includes

Below algorithms are typically used with sorted ranges, though they don't require them

unique unique\_copy

The search algorithms binary\_search, lower\_bound, upper\_bound, and equal\_range require sorted ranges, because they look for values using binary search. Time complicity of these algorithm is O(logn) when we passed random iterator into it.

(such as bidirectional iterators), they still perform only a logarithmic number of

comparisons, but they run in linear time. That's because, lacking the ability to perform

"iterator arithmetic." they need linear time to move from place to place in the range

being searched.

Algorithms set\_union, set\_intersection, set\_difference, and set\_symmetric\_difference offer linear-time performance of the set-theoretical operations their names suggest. Why do they demand sorted ranges? Because without them, they couldn't do their work in linear time.

Unlike the algorithms above discussed, unique and unique\_copy offer well-defined

behaviour even on unsorted ranges. But how it works?

Input elements arranged in such a way, that all duplicate values are next to one another. It eliminates all but the first element from every consecutive group of equal elements.

vector<**int**> v = { 1, 1, 3, 3, 3, 10, 1, 3, 3, 7, 7, 8 };

vector<**int**>::iterator ip;

// Using std::unique

ip = std::unique(v.begin(), v.begin() + 12);

// Now v becomes {1 3 10 1 3 7 8 \* \* \* \* \*} Note: \* means undefined

// Resizing the vector to remove the undefined terms

v.resize(std::distance(v.begin(), ip));

//or, use swap trick :: vector<int>(v).swap(v)

//create the temp object with the help of copy ctor with arg v. Now temp object has only valid //object which is later swapped into vector v.