**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);

}

};