**Item 19. Understand the difference between equality and equivalence.**

find algorithm and set::insert method both determine two objects are same or not. But both perform same operation in different ways.

find's definition of "the same" is equality, which is based on operator== where as set::insert's definition of "the same" is *equivalence*, which is usually based on multiple operators.

We must understand the difference between equality and equivalence to make effective use of the STL.

The notion of equality is based on operator==. If the expression "x == y" returns true, x and y have equal values, otherwise they don't. For Example: two widgets are equal if w1==w2 is true. Means operator == () returns true.

Equivalence is based on the relative ordering of object values in a sorted range. Two objects x and y have equivalent values with respect to the sort order used by an associative container c if neither precedes the other in c's sort order. For example, a set<Widget> s. Two Widgets w1 and w2 have equivalent values with respect to *s* if neither precedes the other in s's sort order. The default comparison function for a set<Widget> is less<Widget>, and by default less<Widget> simply calls operator< for Widgets, so w1 and w2 have equivalent values with respect to operator if the following expression is true:

!(w1 < w2) // it's not true that w1 < w2

&& // and

!(w2<w1) // it's not true that w2 < w1

In simple words neither w1 is less then w2 and nor w2 is greater then w1. This makes sense: two values are equivalent (with respect to some ordering criterion) if neither precedes the others.

The comparison function for an associative container, it's a user-defined predicate. Every standard associative container makes its sorting predicate available through its key\_comp member function, so two objects x and y have equivalent values with respect to an associative container c's sorting criterion if the following evaluates to true:

!c.key\_comp()(x, y) && !c.key\_comp()(y, x)

// it's not true that x precedes y in c's sort order and it's also not true that y precedes x in

// c's sort order.

class LsbLess {

public:

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

return((x % 10) < (y % 10));

}

};

std::set<int, LsbLess> s = { 21,23,26,27 };

std::set<int, LsbLess>::iterator iter1,iter2;

//Algorithm find() looks for equality: if(x==y)

iter1 = find(s.begin(), s.end(), 36); //Iter1 points to s.end()

//set<int>find() looks for equivalence: if ( !(x<y) && !(y<x) )

iter2 = s.find(36); //Iter2 points to 26, but why?

In our example set default comparison operator is overrides by LsbLess. Member method find () check for equivalence not for equality.

(!(x<y)&&!(y<x)) // operator < is replaced by LsbLess

LsbLess(36,26) // equivalent function call return true

// so, Iter2 points to 26.

Not only find () member method in set works on ***equivalence relationship*** but insert also work like same fashion.

For given set s = {21,23,26,27} now want to insert data 46. Even 46 is not present into set, we cannot inter 46 into set s. Because we replace default comparison function with LsbLess which only compare last digit of given number (both 26 & 46 have same last digit).

std::pair<std::set<int>::iterator, bool> ret;

ret = s.insert(46);

if (ret.second == true){

std::cout << "Element inserted\n";

std::cout << \*(ret.first) << "\n";

}

else

std::cout << "Item not inserted\n";

**Another Example of Case-Less String comparison set.**

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.

}

//**Note**:Implementation of *ciStringCompareImpl* is omitted as for now.

set<string, CIStringCompare> ciss; // ciss = "case-insensitive string set”

ciss.insert("Persephone"); // a new element is added to the set

ciss.insert("persephone"); // no new element is added to the set

If we now search for the string "persephone" using set's find member function, the search will succeed.

if (ciss.find("persephone") != ciss.end()) // this test will succeed

but if we use the non-member find algorithm, the search will fail:

if (find( ciss.begin(), ciss.end(),"persephone") != ciss.end()) // this test will fail

That's because "persephone" is *equivalent* to "Persephone" (with respect to the comparison functor CIStringCompare), but it's not equal to it (because string("'persephone") !=string("Persephone")).

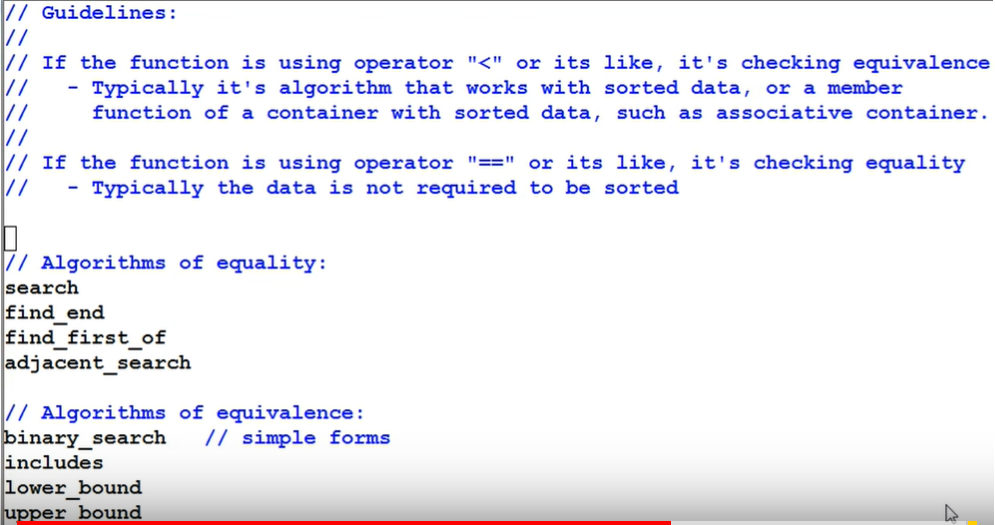
Note: That’s why, prefer member functions (like set::find) to their non-member counterparts (like find).

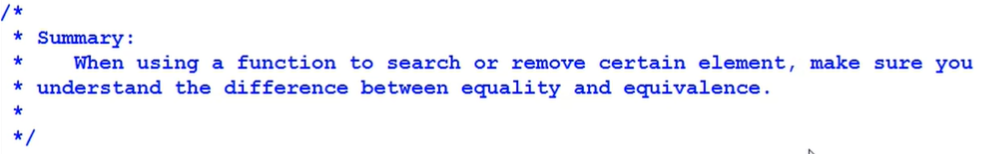
The standard associative containers are kept in sorted order, so each container must

have a comparison function (less, by default) that defines how to keep things sorted.

Equivalence is defined in terms of this comparison function and any standard associative containers has only one comparison function.

If the associative containers used equality to determine when two objects have the same value, then each associative container would require first, comparison function for sorting, a second comparison function for determining when two values are equal.





**Item 20. Specify comparison types for associative containers of pointers.**

set<string\*>ssp; //Set of String pointers.

ssp.insert(new string("Anteater")); //Stored in Add::0xD

ssp.insert(new string("Wombat")); //Stored in Add::0xF

ssp.insert(new string("Lemur")); //Stored in Add::0x1

ssp.insert(new string("Penguin")); //Stored in Add::0x2

set<string\*>::iterator spIter; //String pointer iterator.

for (spIter = ssp.begin(); spIter != ssp.end(); ++spIter) {

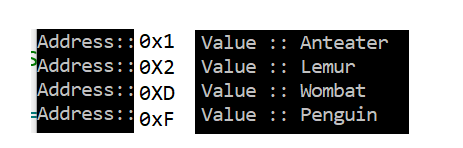
cout << "Address::"<<\*spIter<<" Value :: "<<\*\*spIter << endl;

}

//Set not contained sorted string value instead of that

//Its stored string pointer in sorted order.

Output is:



set<string\*>ssp; //Set of String pointers.

Equivalent to set<string\*, less<string\*>> ssp;

If we want the string\* pointers to be stored in the set in an order determined by the string values, we can't use the default comparison functor class less<string\*>. Instead of that need to write our own comparison functor class, one whose objects take string\* pointers and order them by the values of the strings they point to. Like this:

class StringPtrLess {

public:

bool operator()(const string \*lsh, const string \*rhs)const {

return \*lsh < \*rhs;

}

};

//OR

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

bool operator()(const string \*lsh, const string \*rhs)const {

return \*lsh < \*rhs;

}

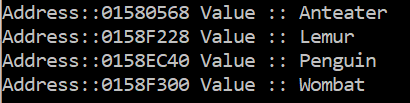
};

set<string\*,StringPtrLess>ssp; //Set of String pointers with custom order

… //insert item into set.

... //Print the items

Output:



**Item 21. Always have comparison functions return false for equal values.**

template<typename T>

class Less {

public:

bool operator()(const T x, const T x) {

return x < y; // If x == y then returns false.

}

};

class LessEq1 {

public:

bool operator()(const T x, const T x) {

return x <= y; // if x==y then returns true, which is

// problematic.

}

};

class LessEq2 {

public:

bool operator()(const T x, const T x) {

if (x == y) //Always return false for equal value.

return false;

else return x < y;

}

};

set<int, Less<int>>s;

No Element; S= {};

s.insert(6);

Insert 6; S= {6};

s.insert(4);

Insert 4; S= {6,4};

How it determines element 4 is present in set or not. It will check for equivalency based on below expression:

!(4<6)&&!(6<4)

In such case any of comparator function return true, resultant final expression output false.

***Note****: If output is false means elements are not equivalent (hence not same) goes directly into the Set.*

6

/

4

s.insert(2);

s.insert(2);

//This time both of comparator function return false.

!(w1<w2) && !(w2<w1) Expression used to test equivalency.

Where w1<w2 and w2<w1 are the Boolean

output by comparator function Less.

!false && !false => true Which means the '2' which we are

going to insert, is equivalent to

another '2' which is pre-existed in

container.

Suppose we have another comparator function LessEq1 <=

set<int, LessEq1<int>> s1;

s1.insert(10); //10a

s1.insert(10); //10b

Now our container will be crashed after inserting second copy of 10. But Why?

This time comparator function LessEq1 return true for equal values

!(10 LessEq1 10) && !(10 LessEq1 10) = false.

Now the above express output is false which means both copies of 10’s are not

equivalent. When it tries to insert second copy of 10's the container itself corrupted, because set only holds the unique value.

set<int,LessEq2<int>> s2;

s2.insert(10);

s2.insert(10);

Now LessEq2 gives false for equal values. And expression output true that

means 10 is already present in set container don’t add another one.

//Even in multiset which contains equal value, also comparator

//function return false for equal value.

multiset<int, LessEq1<int>> ms1;

ms1.insert(10);

ms1.insert(10);

Second 10 insertion will corrupt the container. Why??

Before preceding to why above is failed let’s consider small example:

--------------------------------------------------------------------------------------------

vector<int> v = { 10,10,30,30,30,100,10,300,300,70,70,80};

// sort the vector with user defined comparator function

sort(v.begin(), v.end(), Less<int>());

// for descending order sort only replace the operator < with op >

pair<vector<int>::iterator, vector<int>::iterator> piter;

piter = equal\_range(v.begin(), v.end(), 30);

// Displaying the subrange bounds

cout << "10 is present in the sorted vector from index "

<< (piter.first - v.begin()) << " till " << (piter.second -

v.begin());

---------------------------------------------------------------------------------------------

Now come to the original problem, why multiset not accepting two copy of equivalent value? Here the comparator function returns *true* and equivalency expression output is *false*. Which means both copies of 10’s are not equivalent.

Suppose if second copy of 10 inserted into map, then how equal\_range() method

behave on this container? Because equal\_range() assume both 10's are same but

container says that both are different (not equivalent), and It is a contradiction. So, better solution is to prevent second copy from insertion into container.

// Now again we create multiset with comparator function LessEq2,

// which return false when both values are equal, this time it will

// work. Container understand both copy of 10’s are equivalent.

multiset<int, LessEq2<int>> ms1;

ms1.insert(10);

ms1.insert(10);

***Moral: Always return false for equal value form user defined comparator.***

**Item 22. Avoid in-place key modification in set and multiset.**

class Employee {

public:

Employee(const Employee &e):name(e.name),id(e.id){}

Employee(const string &nm,int \_id):name(nm),id(\_id){}

const string& getName()const {

return name;

}

void setName(const string &name) {

this->name = name;

}

int idNumber() const { return id; }

void SetID(int x){ id=x; }

private:

string name;

int id;

};

void showEmpData(const Employee &e) {

cout << e.getName().c\_str() << " " << e.idNumber() << endl;

}

struct EmpIdCompare : public binary\_function <Employee, Employee, bool> {

bool operator()(const Employee &lsh, const Employee &rhs)const {

return lsh.idNumber() < rhs.idNumber();

}

};

map<char, int> alphabetMap;

pair<char, int> p;

p = make\_pair('A', 65);

alphabetMap.insert(p);

p = make\_pair('B', 66);

alphabetMap.insert(p);

alphabetMap.insert({ 'C',67 });

map<char, int>::iterator mapIter = alphabetMap.begin();

//mapIter->first = 'Z'; // error! map keys can't be

//changed,in-palce modification not allowed

mapIter->second = 100; // Non-Key part we can change.

// That's because the elements in an object of type map<K, V> or multi-

// map<K, V> are of type pair<const K, V>. Because the type of the key is

// const K.it can't be changed.

set<Employee, EmpIdCompare> setEmp;

setEmp.insert(Employee("Rajeev", 10));

setEmp.insert(Employee("Bhautik", 2));

setEmp.insert(Employee("Omi", 17));

setEmp.insert(Employee("Kabil", 5));

setEmp.insert(Employee("Gaukul", 9));

set<Employee, EmpIdCompare>::iterator setIter =

setEmp.find(Employee("Kabil", 5));

if (setIter != setEmp.end()) {

showEmpData(\*setIter); //Iter has always const ref.

const\_cast<Employee&>(\*setIter).setName("KabilaSaran"); // Convert const to non-const Employee, because \*i is const

// const\_cast<Employee&>(\*setIter).SetID(2);

// Don’t alter the key part of set. it will corrupt your container.

// same ID (2) present multiple time.

}

setEmp.insert(Employee("Dinesh", 3));

setEmp.insert(Employee("Kumar", 3)); //Not allowd to inter, because key alread exist.

for\_each(setEmp.begin(), setEmp.end(), showEmpData);

//Modification of set non-key data is legal, but bit complicated because

//it required const-casting. Better approch is:

//1. Locate the object in container.

setIter = setEmp.find(Employee("Kabil", 5));

//2. Make a copy of the element to be modified.

Employee tmp(\*setIter);

//3. Delete employee form container.make sure iter++

setEmp.erase(setIter++);

//4. Modifiy the element

tmp.setName("Sharma");

//5. Add object into container.

setEmp.insert(tmp);

//setEmp.insert(setIter, tmp); // Or, Iter as a help

**Item 23. Consider replacing associative containers with sorted**

**vectors.**

/\*

Sometimes we required data structure offering fast lookups, immediately think of the standard associative containers, set, multiset, map, and multimap which offers lookup in logarithmic time. Also, we have nonstandard hashed containers which offers constant time for lookup (With poorly chosen hashing functions or with table sizes that are too small, the performance of hash table lookups may degrade significantly).

Choosing different container depends upon below point:

1. Setup. Create a new data structure by inserting lots of elements into it. During this phase, almost all operations are insertions and erasures. Lookups are rare or non-existent.

2. Lookup. Consult the data structure to find specific pieces of information.

During this phase, almost all operations are lookups. Insertions and erasures

are rare or non-existent.

3. Reorganize. Modify the contents of the data structure, perhaps by erasing all the current data and inserting new data in its place. Behaviourally, this phase is equivalent to phase 1. Once this phase is completed, the application returns to phase 2.

When we required only lookup, have batter choice of vector rather than any associative containers. Implementation of standard associative containers demands more memory then vector.

\*/

class Employee {

public:

Employee(const Employee &e) :name(e.name), id(e.id) {}

Employee(const string &nm, int \_id) :name(nm), id(\_id) {}

const string& getName()const {

return name;

}

int idNumber() const { return id; }

private:

string name;

int id;

};

typedef pair<string, string> airportCodeMap;

void showEmpData(const Employee &e) {

cout << e.getName().c\_str() << " " << e.idNumber() << endl;

}

struct EmpIdCompare : public binary\_function <Employee, Employee, bool> {

bool operator()(const Employee &lsh, const Employee &rhs)const {

return lsh.idNumber() < rhs.idNumber();

}

};

struct AirportCodeMapCompare :public binary\_function<airportCodeMap, airportCodeMap, bool> {

bool operator()(const airportCodeMap &c1, const airportCodeMap &c2)const {

return keyCompare(c1.first, c2.first);

}

// We can't know whether the key value or the pair will be passed as the first // argument or second one. So, need to write two different comparison functions // for lookups: one where the key value is passed first and one where the pair // is passed first.

bool operator()(const airportCodeMap &c1, const

airportCodeMap::first\_type &key) {

return keyCompare(c1.first, key);

}

bool operator()(const airportCodeMap::first\_type &key, const

airportCodeMap &c2) {

return keyCompare(key,c2.first);

}

private:

bool keyCompare(const airportCodeMap::first\_type &s1, const

airportCodeMap::first\_type &s2)const {

return s1 < s2;

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

s2.end(), ciCharLess);

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

s2.end(), [](int x, int y) {return x < y;});

}

};

void airportCodeMapDisplay(const airportCodeMap &codeMap){

cout << "Code:: " << (codeMap.first).c\_str() << ", Name:: " <<

(codeMap.second).c\_str() << endl;

}

vector<Employee>vEmp;

// Alternative to set<Employee> or multiset<Employee> with

// vector<Employee>.

// 1: Setup phase: lots of insertions, few lookups

vEmp.push\_back(Employee("Rajeev", 10));

//Vector can allow dulpicate entry to behave like mutiset.

vEmp.push\_back(Employee("Rajeev", 10));

vEmp.push\_back(Employee("Rajeev", 10));

vEmp.push\_back(Employee("Bhautik", 2));

vEmp.push\_back(Employee("Omi", 17));

vEmp.push\_back(Employee("Kabil", 5));

vEmp.push\_back(Employee("Gaukul", 9));

// End of Setup phase. When simulating a multiset, we might prefer

// stable\_sort instead.

for\_each(vEmp.begin(), vEmp.end(), showEmpData);

stable\_sort(vEmp.begin(), vEmp.end(), EmpIdCompare());

// Now we can perform any lookup method which required sorted range and

// offers logritmic

// time operation: binary\_serach, upper\_bound, lower\_bound,

// euqal\_range(not required sorted data)

// ...

//End of lookup phase, start Reorganize phase.

stable\_sort(vEmp.begin(), vEmp.end(), EmpIdCompare());

// Alternative to map<char,int> or multimap<char,int> with

// vector<pair<char,int>>. map and multimap has pair of <K,V> more

// precisely map<const K,v> and multimap<K,V>. When using a vector to

// emulate a map<K, V>, then, the type of the data stored

// in the vector will be pair<K, V>, not pair<const K, V>. Because need

// to vector sort which required move key value one to another place.

vector<airportCodeMap> codeMapVector;

codeMapVector.push\_back(make\_pair<string, string>("CCU", "Kolkata"));

codeMapVector.push\_back(make\_pair<string, string>("PAT", "Patna"));

codeMapVector.push\_back(make\_pair<string, string>("BLR", "Bangalore"));

codeMapVector.push\_back(make\_pair<string, string>("RNC", "Ranchi"));

codeMapVector.push\_back(make\_pair<string, string>("NDLS", "New Delhi"));

codeMapVector.push\_back(make\_pair<string, string>("HYD", "Hydrabad"));

codeMapVector.push\_back(make\_pair<string, string>("MUB", "Mumbai"));

codeMapVector.push\_back(make\_pair<string, string>("CCN", "Chennai"));

sort(codeMapVector.begin(), codeMapVector.end(),

AirportCodeMapCompare());

//Look-up phase:

if (binary\_search(codeMapVector.begin(), codeMapVector.end(), "RNC",

AirportCodeMapCompare())) {

cout << "data found\n";

}

**Item 24. Choose carefully between map::operator[] and map-insert when efficiency is important.**

//See the STL copy.

**Item 25. Familiarize yourself with the nonstandard hashed containers.**

**Nonstandard associative containers :**hash\_set, hash\_multiset, hash\_map, and hash\_multimap.

In the C++ programming language, **Nonstandard associative containers,** **is the name of a hashed associative container in the Standard Template Library**. It is provided by several implementors, such as the GNU C++ compiler and Microsoft's Visual C++. **It is not part of the C++ Standard Library**, but the C++ Technical Report 1(TR1) contains the very similar container unordered\_map, which will be included in the upcoming C++0x standard.

So, in short,

* **YES,** it's part of the STL.
* **But it IS NOT** part of the standard library.
* But it is supported by several very popular implementations.