Strict_weak_ordering_and_stl.md - Grip

Did you compare it right?

If you had ever used a map or set or even std::sort I bet you would have to give a comparator function. (Or an overload to the < (less than) operator). I will try to give an overview of how certain associative stl containers use this property for ordering the elements. Almost all stl containers rely on strict weak ordering A strict weak ordering defines the relative position of elements in terms of precedence of one item over other. For eg. if you have a room full of person and you have to form a queue based on their height, a person with "lesser" height will "precede" the person with greater height. For a function to be satisfying strict weak ordering following conditions need to be met:

- `a < b => !(b < a)`
- `!(a < b) && !(b < a) => a≡b`

Note that the "≡" sign denotes **equivalence** which can be quite different from equality. Item 19 of "Effective Stl by Scott Meyers" is often quoted as a good source for understanding the difference.

To give an intution of how the stl containers use the comparator function, lets take the example of simple binary search.

```
template <typename T>
size_t binary_search(const container<T> &a, const T &key, std::function<bool(const T &, const T&)> less){
    size_t high = a.size(),low = 0;
    while(low < high){
        size_t mid = low + ((high - low) >> 1);
        if (!less(a[mid], key) && !less(key, a[mid])) return mid;
        else if (less(a[mid], key)) low = mid + 1;
        else high = mid;
    }
    return a.size(); // failure condition
}
```

This line is equivalent to a≡b

```
        if (!less(a[mid], key) && !less(key, a[mid]))
```

This is sort of similar to how `map::find` and `set::find` would work.

What could go wrong

This brings us to the main theme of this white paper. So it should be evident now that the comparator function that the stl methods expect should be **strict weak ordered**. Like all other apis once you break the contract you are in undefined behavior land. Can you figure out what's wrong in this code?

```
vector<int> a{0,0,0,0};
cout << std::binary_search(a.begin(), a.end(),0 std::less_equal<int>);
```

binary_search if called this way will return `false` even though 0 is present in the vector! If you try to do a dry run with the example of binary code presented earlier you will realize why. But just to state it down mathematically the condition that `a < b => !(b < a)` is violated when `a==b` when you use less_equal instead of less. This was more of a hypothetical scenario. Lets take an example which I saw recently

A code like this

```
void sortReverse(vector<int> values){
    sort(values.begin(),values.end(),     std::greater_less<int>());
}
```

can cause a crash or even a hang depending on the internal implementation of sort. For eg, when calling this function in MSVC(VS 2015) and g++(4.8) like this

```
vector<int> v(50, 0);// 50 elements each filled with 0
sortReverse(v);
```

caused an access violation/segmentation fault. Needless to say how giving a wrong implementation of the comparator function can not only lead to wrong behavior it can also leave ***security loopholes in your code.***!!!

Correctly implementing the < operator

The only perfect way to be sure that your comparator function is valid is to mathematically prove that it meets the requirement of strict weak ordering. Obviously this can be a pain specially in the era where the focus is to churn out code quickly. ;) One of the tricks that I felt very neat is using the std::tie method to create a std::tuple and then use it's < operator which is known to behave correctly. for eg:

```
struct Person{
int height;
int age;
string name;
};
```

To write a correct comparator function one will require at least three if's. Instead this could be written using tie as

```
bool operator<(const Person &a, const Person &b)
{
    return tie(a.height,a.age,a.name) < tie(b.height, b.age, b.name);
}
```

and done :) You have effectively shifted the responsibility of proving the correctness of your code to std::tuple's code.

References:

- Infinite loop while sorting
- Memory corruption while sorting
- Doesnt stl sort require a strict weak ordering to work?