C++ STL MAP WITH MULTIPLE KEYS*

Brian West

Abstract

This article shows to get a STL map to work with multiple keys. Instead of having a single key mapped to a single value, a STL map will be shown that maps multiple keys to a single value. Two ways to accomplish will be shown.

The C++ STL map is an associative map between a single key and a single value. In addition, the key must be unique for the given map. The C++ STL also provides a multimap template, which removes the unique key restriction. A single key in a multimap can map to multiple values. The STL multimap name might confuse the gentle reader into believing that it supports a map with multiple keys, which it does not. A map with multiple keys can be realized with the STL map as shown below.

Most examples of the C++ STL map show a single key mapping to a single value. The map key is usually shown as a simple data type like an integer or a string, but the key also can be an aggregate object. In particular, the key object can be a struct/class containing multiple keys. For example,

```
class MultiKey {
  public:
    long  key1;
    int   key2;
    long  key3;
    long  key4;

    MultiKey(long k1, int k2, long k3, long k4)
      : key1(k1), key2(k2), key3(k3), key4(k4) {}
};
```

Figure 1: Multi-key container

Naively, a multi-key map instance using the above definition can be constructed via

```
std::map<MultiKey, int> mymap;
```

*Version 1.2: Oct 22, 2010 8:30 pm -0500
†http://creativecommons.org/licenses/by/3.0/

The gentle reader will discover that the above C++ snippet will fail to compile. The missing magic is that a "less" comparator function/method (aka the less than operator) needs to be supplied.

Requiring a the less than operator is a bit counter intuitive. The STL map must be able to compare two keys for equality, to maintain key uniqueness when adding to the map. The equality operator can be defined via the less than operator, as will be shown below.

Internally the STL map is sorted by the key via the less than operator. Sorting makes determining membership efficient. Without key sorting, determining membership would require accessing all map entries, worst case. With key sorting and an appropriate data structure like a btree, the membership becomes more efficient.

There are multiple ways to add the less than operator to the above example. Adding a `operator<()` to the above class is the most straight forward.

```
class MultiKey {
  public:
    long  key1;
    int   key2;
    long  key3;
    long  key4;

    MultiKey(long k1, int k2, long k3, long k4)
      : key1(k1), key2(k2), key3(k3), key4(k4) {}

    bool operator<(const MultiKey &right) const
    {
        if ( key1 == right.key1 ) {
            if ( key2 == right.key2 ) {
                if ( key3 == right.key3 ) {
                    return key4 < right.key4;
                }
                else {
                    return key3 < right.key3;
                }
            }
            else {
                return key2 < right.key2;
            }
        }
        else {
            return key1 < right.key1;
        }
    }
};

std::map<MultiKey, int> mymap;
```

Figure 2: Multi-key container with less than operator

Note the implementation of `operator<()`. The individual keys are given a precedence while checking for less than. A straight forward but incorrect implementation of less than looks like this:

```
bool operator<(const MultiKey &right) const
{
    return ( key1 < right.key1; && key2 < right.key2; &&
             key3 < right.key3; && key4 < right.key4; );
}
```

Figure 3: Straight forward but incorrect

The above code fails when comparing the results of `multikey1 < multikey2` and `multikey2 < multikey1`. This failure also will cause problems when checking for key equality in the membership test. As mentioned above, the less than operator is used to make the equality calculation. The equality operator is implemented something like this:

```
bool operator==(const MultiKey &right) const
{
   return ( !(this < right) && !(right < this) );
}
```

Figure 4: Pseudo code for the equality operator

Thus, multi-keys are equal when neither is less than the other.

An alternative less than operator

When declaring a map instance, the less than operator can be explicitly provided like this:

```
    class MultiKey {
  public:
    long  key1;
    int   key2;
    long  key3;
    long  key4;

    MultiKey(long k1, int k2, long k3, long k4)
      : key1(k1), key2(k2), key3(k3), key4(k4) {}

    bool operator<(const MultiKey &right) const
    {
        if ( key1 == right.key1 ) {
            if ( key2 == right.key2 ) {
                if ( key3 == right.key3 ) {
                    return key4 < right.key4;
                }
                else {
                    return key3 < right.key3;
                }
            }
            else {
                return key2 < right.key2;
            }
        }
        else {
            return key1 < right.key1;
        }
    }
};

struct multikey_less : public std::binary_function<MultiKey, MultiKey, bool>
{
    bool
    operator()(const MultiKey &mkey1, const MultiKey &mkey2 ) const
    {
        return mkey1.operator<(mkey2); // mkey1 < mkey2;
    }
};

std::map<MultiKey, int, multikey_less> mymap2;
```

Figure 5: An alternative less than

I did not want to re-implement the class MultiKey so multikey_less just calls MultiKey's less than operator. Obviously, if the less than operator existed and it did what you needed, there would be no need for

creating multikey_less. If the existing less than operator is not what is needed though, it can be overridden via a less than comparator like multikey_less above (but with an independent implementation), supplied on the instance declaration.