# Learning C++: The STL and Maps and Multimaps

There are many applications that require data to be associated with other data in a key-value relationship. Dictionaries, phone lists, and inventories are just three examples. The primary container in the Standard Template Library (STL) for storing associative data is the map class. This class allows the storage of unique keys and their associated values. Associated with the map class is the multimap class that allows duplicate keys. In this article I'll show you how to use maps and multimaps for working with associative data.

## Fundamentals of the map and multimap Classes

The map and multimap classes are implemented using balanced binary trees. Each key/value pair is a node in the tree and the nodes are placed into the tree based on the key value. This means that maps (and multimaps) are very well sorted and it is very efficient to search for a key in order to discover the value associated with it. This structure also means, however, that is very inefficient to search for a value without providing the key and isn't even really implemented in the class.

Another inefficient operation on a map or multimap is changing a key. You can't do this directly; you must delete the key/value pair and then insert a new pair with the modified key. This is because modifying the key in place can make the internal binary tree unbalanced and the tree must rebalance itself as part of the modification operation. This operation is so inefficient, in fact, that it is not allowed.

## Creating Maps

The preprocessor directive to enter for using a map is:

#include <map>

A map is a template class that requires template parameter arguments for the key and the value, in that order. Here are some examples:

map<string, string> phoneList;

map<string, int> birthYears;

map<int, int> grades; // where first int is id number

You can provide a third template parameter if you want to modify the sort order of the map:

map<int, int, greater<int>> grades;

You can provide an initializer list to initialize a map with some key/value pairs:

map<string, int> grades = {{"Doe", 81}, {"Brown", 92},

{"Smith", 88}};

## A Quick Review of the pair Structure

You may have noticed in the initializer list above that the key/value pairs were entered in the form *{key, value}*. This form is used to delineate a pair, which is a structure found in the utility library. The pair structure has two fields, first and second. Here is how you can create a pair outside of a map:

#include <iostream>

#include <utility>

using namespace std;

int main()

{

pair<string, int> student;

student.first = "Brown";

student.second = 88;

cout << "Name: " << student.first << ", Grade: "

<< student.second << endl;

return 0;

}

The output from this program is:

Name: Brown, Grade: 88

There are some instances when you will use these pair functions directly when working with maps so it's a good idea to understand how pairs work.

## Adding Data to a Map

The insert function is used to add a new element to a map:

#include <iostream>

#include <map>

using namespace std;

int main()

{

map<string, int> grades = {{"Doe", 81}, {"Brown", 92},

{"Smith", 88}};

grades.insert({"Jones", 79});

return 0;

}

I won't provide a complete program again unless one of the preprocessor directives changes.

The insert function does return a pair as its return value, containing an iterator pointing to the insertion point in the first field and a boolean representing success or failure of the insertion in the second field. Here is an example:

#include <iostream>

#include <map>

#include <utility>

using namespace std;

int main()

{

map<string, int> grades = {{"Doe", 81}, {"Brown", 92},

{"Smith", 88}};

pair<map<string, int>::iterator, bool> success =

grades.insert({"Jones", 79});

if (success.second) {

cout << "inserted at: " << success.second << endl;

}

return 0;

}

## Accessing Map Data

If you trying to access the value associated with a specific key, you can either use the [] operator and put the key inside the brackets or use the at function with the key as the argument. Here's an example using both methods:

int main()

{

map<string, int> grades = {{"Doe", 81}, {"Brown", 92},

{"Smith", 88}};

grades.insert({"Jones", 79});

string name = "Brown";

cout << name << "'s grade: " << grades[name] << endl;

name = "Doe";

cout << name << "'s grade: " << grades.at(name) << endl;

return 0;

}

The output from this program is:

Brown's grade: 92

Doe's grade: 81

If you want to access all the elements of a map you can use a range for loop, like this:

int main()

{

map<string, int> grades = {{"Doe", 81}, {"Brown", 92},

{"Smith", 88}};

grades.insert({"Jones", 79});

for(auto iter = grades.begin(); iter != grades.end();

iter++) {

cout << iter->first << ": " << iter->second << endl;

}

return 0;

}

Each iteration returns an iterator pointing to a pair structure. To get the key, you must reference the first field and to get the value you must reference the second field. And to access these fields, you must use the arrow operator -> rather than the dot operator.

## Removing Elements from a Map

You remove map elements using the erase function. This function takes a key as an argument and removes the element with a matching key. The function returns the number of elements removed.

Here is an example:

int main()

{

map<string, int> grades = {{"Doe", 81}, {"Brown", 92},

{"Smith", 88}};

string key = "Doe";

int numRemoved = grades.erase(key);

cout << "Removed " << numRemoved << " elements." << endl;

return 0;

}

## Using Multimaps

A multimap is just a map that allows duplicate keys. Sometimes that's the right container to use, such as when you are building a dictionary or a phone list where people on the list can have the same name. The following program fragment will demonstrate how to use a multimap to create a dictionary where the same word can have multiple meanings:

multimap<string, string> dictionary;

dictionary.insert({"bark", "outer covering of a tree"});

dictionary.insert({"bark", "sound a dog makes"});

dictionary.insert({"nail", " as in toenail, fingernail"});

dictionary.insert({"nail", "sharp sliver of metal"});

dictionary.insert({"mine", "possesive adjective"});

dictionary.insert({"mine", "place where minerals are located"});

We can traverse the complete dictionary using a range for loop:

for (const auto elem : dictionary) {

cout << elem.first << ": " << elem.second << endl;

}

Now let's traverse just the definitions for one word. To do that I need to introduce two new functions, lower\_bound and upper\_bound. These functions take a key as an argument. The lower\_bound function returns the first position where a new element would get inserted or the position of where the first element that matches the key is found. The upper\_bound function returns the last position where a new element would get inserted or the position of where the last element that matches the key is found.

We can use these functions to return all the definitions of a word, as in this example:

//find all definitions of one word

string key("mine");

cout << "All definitions of mine: " << endl << endl;

for (auto iter = dictionary.lower\_bound(key);

iter != dictionary.upper\_bound(key); iter++) {

cout << iter->first << ": " << iter->second << endl;

}

The output from this program fragment is:

All definitions of the word mine:

mine: possesive adjective

mine: place where minerals are located

## Using Maps and Multimaps

The map and multimap containers are specialized containers that should only be used when you are working with associative data. The map class is used when the keys in your application need to be unique and the multimap class is used when the keys in the application can be duplicates. These containers are also sorted so you can change the sort criteria if you want your keys to appear in an order that isn't ascending.

Thanks for reading this article and please email me with comments and suggestions.