

Starting out

[Get the Ebook](#)
[Get Started with C or C++](#)
[Getting a Compiler](#)
[Book Recommendations](#)

Tutorials

[C Tutorial](#)
[C++ Tutorial](#)
[Java Tutorial](#)
[Game Programming](#)
[Graphics Programming](#)
[Algorithms & Data Structures](#)
[Debugging](#)
[All Tutorials](#)

Practice

[Practice Problems](#)
[Quizzes](#)

Resources

[Source Code](#)
[Source Code Snippets](#)
[C and C++ Tips](#)
[Finding a Job](#)

References

[Function Reference](#)
[Syntax Reference](#)
[Programming FAQ](#)

Getting Help

[Message Board](#)
[Ask an Expert](#)
[Email](#)

STL Maps -- Associative Arrays



By Alex Allain

Suppose that you're working with some data that has values associated with strings -- for instance, you might have student usernames and you want to assign them grades. How would you go about storing this in C++? One option would be to write your own **hash table**. This will require writing a hash function and handling collisions, and lots of testing to make sure you got it right. On the other hand, the standard template library (STL) includes a templated class to handle just this sort of situation: the STL map class, which conceptually you can think of as an "associative array" -- key names are associated with particular values (e.g., you might use a student name as a key, and the student's grade as the data).



In fact, the STL's map class allows you to store data by any type of key instead of simply by a numerical key, the way you must access an array or vector. So instead of having to compute a hash function and then access an array, you can just let the map class do it for you.

To use the map class, you will need to include `<map>` and maps are part of the std **namespace**. Maps require two, and possibly three, types for the template:

Decisions,

[learn more](#)

birst
Cloud BI & Analytics

```
std::map <key_type, data_type, [comparison_function]>
```

Notice that the comparison function is in brackets, indicating that it is optional so long as your *key_type* has the less-than operator, *<*, defined -- so you don't need to specify a comparison function for so-called primitive types such as *int*, *char*, or even for the *string* class. Moreover, if you overloaded the *<* operator for your own class, this won't be necessary.

The reason that the key type needs the less-than operator is that the keys will be stored in sorted order -- this means that if you want to retrieve every key, value pair stored in the map, you can retrieve them in the order of the keys.

Let's go back to the example of storing student grades. Here's how you could declare a variable called *grade_list* that associates strings (student names) with characters (grades -- no + or - allowed!).

```
std::map <string, char> grade_list;
```

Now, to actually store or access data in the map, all you need to do is treat it like a normal array and use brackets. The only difference is that you no longer use integers for the index -- you use whatever data type was used for the key.

For instance, following the previous example, if we wanted to assign a grade of 'B' to a student named "John", we could do the following:

```
grade_list["John"] = 'B';
```

If we later decided that John's grades had improved, we could change his grade in the same fashion:

```
grade_list["John"] = 'B';  
// John's grade improves  
grade_list["John"] = 'A';
```

So adding keys to a map can be done without doing anything special -- we just need to use the key and it will be added automatically along with the corresponding data item. On the other hand, getting rid of an element requires calling the function *erase*, which is a member of the map class:

```
erase(key_type key_value);
```

For instance, to erase "John" from our map, we could do the following:

```
grade_list.erase("John");
```

If we're curious, we could also check to see how many values the map contains by using the *size* function, which is also a member of the map class and other containers:

```
int size();
```

For instance, to find the size of our hypothetical class, we could call the *size* function on the *grade_list*:

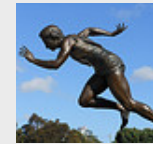
Popular pages



[C Tutorial](#)



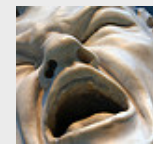
[Exactly how to get started with C++ \(or C\) today](#)



[5 ways you can learn to program faster](#)



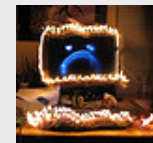
[C++ Tutorial](#)



[The 5 Most Common Problems New Programmers Face](#)



[How to make a game in 48 hours](#)



[8 Common Programming Mistakes](#)



What is C++11?

Image credits

```
std::cout<<"The class is size "<<grade_list.size()<<std::endl;
```

If we're only interested in whether the map is empty, we can just use the map member function `empty`:

```
bool empty();
```

`empty` returns true if the map is empty; it returns false otherwise.

If you want guarantee that the map is empty, you can use the `clear` function. For instance:

```
grade_list.clear();
```

would remove all students from the `grade_list`.

What if you just wanted to check to see whether a particular key was in the map to begin with (e.g., if you wanted to know whether a student was in the class). You might think you could do something like this using the bracket notation, but then what will that return if the specified key has no associated value?

Instead, you need to use the `find` function, which will return an **iterator** pointing to the value of the element of the map associated with the particular key, or if the key isn't found, a pointer to the iterator `map_name.end()`! The following code demonstrates the general method for checking whether a key has an associated value in a map.

```
std::map<string, char> grade_list;
grade_list["John"] = 'A';
if(grade_list.find("Tim") == grade_list.end())
{
    std::cout<<"Tim is not in the map!"<<endl;
}
```

Iterators can also be used as a general means for accessing the data stored in a map; you can use the basic technique from before of getting an iterator:

```
std::map<parameters>::iterator iterator_name;
```

where *parameters* are the parameters used for the container class this iterator will be used for. This iterator can be used to iterate in sequence over the keys in the container. Ah, you ask, but how do I know the key stored in the container? Or, even better, what exactly does it mean to dereference an iterator over the map container? The answer is that when you dereference an iterator over the map class, you get a "pair", which essentially has two members, first and second. First corresponds to the key, second to the value. Note that because an iterator is treated like a pointer, to access the member variables, you need to use the arrow operator, `->`, to "dereference" the iterator.

For instance, the following sample shows the use of an iterator (pointing to the beginning of a map) to access the key and value.

```
std::map<string, char> grade_list;
grade_list["John"] = 'A';
// Should be John
std::cout<<grade_list.begin()->first<<endl;
```

```
// Should be A
std::cout<<grade_list.begin()->second<<endl;
```

Finally, you might wonder what the cost of using a map is. One issue to keep in mind is that insertion of a new key (and associated value) in a map, or lookup of the data associated with a key in a map, can take up to $O(\log(n))$ time, where n is the size of the current map. This is potentially a bit slower than some hash tables with a good hashing function, and is due to the fact that the map keys are stored in sorted order for use by iterators.

Summary

The Good

- Maps provide a way of using "associative arrays" that allow you to store data indexed by keys of any type you desire, instead of just integers as with arrays.
- Maps can be accessed using iterators that are essentially pointers to templated objects of base type pair, which has two members, first and second. First corresponds to the key, second to the value associated with the key.
- Maps are fast, guaranteeing $O(\log(n))$ insertion and lookup time.

The Gotchas

- Checking for whether an element is in a map requires using the find function and comparing the returned iterator with the iterator associated with the end of the map.
- Since maps associate a key with only one value, you need to use a multimap (or make the type of the data a container) if you must associate multiple pieces of data with one key.

[Previous: Iterators in the STL](#)

[Next: Learn about STL Linked Lists](#)

Related articles

[Read about the vector container class](#)



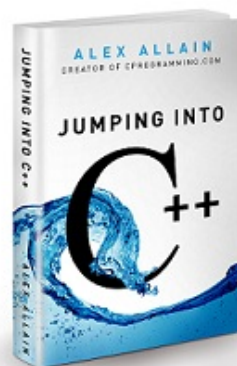
25



3



9




Want to become a C++ programmer? The Cprogramming.com ebook, *Jumping into C++*, will walk you through it, step-by-step. Get *Jumping into C++* today!


Popular pages

- [Exactly how to get started with C++ \(or C\) today](#)
- [C Tutorial](#)
- [C++ Tutorial](#)
- [5 ways you can learn to program faster](#)
- [The 5 Most Common Problems New Programmers Face](#)
- [How to set up a compiler](#)
- [8 Common programming Mistakes](#)
- [What is C++11?](#)
- [How to make a game in 48 hours](#)

Recent additions

- [How to create a shared library on Linux with GCC](#) - December 30, 2011
- [Enum classes and nullptr in C++11](#) - November 27, 2011
- [Learn about The Hash Table](#) - November 20, 2011
- [Rvalue References and Move Semantics in C++11](#) - November 13, 2011
- [C and C++ for Java Programmers](#) - November 5, 2011
- [A Gentle Introduction to C++ IO Streams](#) - October 10, 2011



AdChoices 

[► Key Programming](#)

[► Template C++](#)

[► Array](#)

[► C++ Iterators](#)

Join our mailing list to keep up with
the latest news and updates about
Cprogramming.com!



Name

Email

Sign up!

236323 READERS
BY AWEBER

[Advertising](#) | [Privacy policy](#) | Copyright © 1997-2011 Cprogramming.com. All rights reserved. |
webmaster@cprogramming.com