Stephen Welch

START LEARNING

Looking to learn more about maps in C++ and how to use them? Then this article is for you. We cover the details on how exactly to use maps, present some use cases for them and explain when to avoid them entirely. Let’s dive in.

## What is a map in C++?

A C++ map is a way to store a key-value pair. A map can be declared as follows:

```
#include <iostream>
#include <map>

map<int, int> sample_map;
```

Each map entry consists of a pair: a key and a value. In this case, both the key and the value are defined as integers, but you can use other types as well: strings, vectors, types you define yourself, and more.

## C++ map use cases

There are two main reasons why the map type can be valuable to C++ developers. First, a map allows fast access to the value using the key. This property is useful when building any kind of index or reference. Second, the map ensures that a key is unique across the entire data structure, which is an excellent technique for avoiding duplication of data.
By virtue of these two advantages, a map is a common choice if you’re developing, for instance, a trading application in which you need to store stock prices by ticker symbol. If

## How do I use a map in C++?

The primary operations you'll perform with a map are creating a new map, adding elements to and reading elements from a map, and iterating through every element in a map. Let's take a look at each of these actions.

### Constructing a map

There are five ways to construct a map in C++, but two of them are much more commonly used than the others. The first way is to create an empty map, then add elements to it:

```
#include <map>
#include <string>
using namespace std;

int main() {
  map<int, string> sample_map;
  sample_map.insert(pair<int, string>(1, "one"));
  sample_map.insert(pair<int, string>(2, "two"));

  cout << sample_map[1] << " " << sample_map[2] << endl;
}
```

In this example, we create a map that uses integers as keys and strings as values. We use the pair construct to create a key-value pair on the fly and insert that into our map. The second often-used option is to initialize the map to a list of values at declaration. This option has been available since the C++11 standard and therefore isn't supported by older compilers, but it allows for clearer declaration:

```
#include <iostream>
#include <map>
#include <string>
```

```
  cout << sample_map[1] << " " << sample_map[2] << endl;
}
```

Other ways to create a map include copying an existing map, copying parts of an existing map (by indicating a start position and an end position for the copy) and moving elements from another map without creating an intermediate copy.

### Accessing map elements

In order to access the elements of the map, you can use array-style square brackets syntax:

```
...
cout << sample_map[1] << endl;
...
```

Another option, available as of C++11, is the at method:

```
...
cout << sample_map.at(1) << endl;
...
```

## Inserting elements

When inserting elements into a map, it's common to use either the square brackets syntax or the insert method:

```
...
sample_map.insert(pair<int, string>(4, "four"));
```

In some cases, you might need to walk through a map and retrieve all the values in it. You can do this by using an *iterator*—a pointer that facilitates sequential access to a map's elements.

An iterator is bound to the shape of the map, so when creating an iterator you'll need to specify which kind of map the iterator is for. Once you have an iterator, you can use it to access both keys and values in a map. Here's what the code would look like:

```
int main() {
  map<int, string> sample_map { { 1, "one"}, { 2, "two" } };
  sample_map[3] = "three";
  sample_map.insert({ 4, "four" });

  map<int, string>::iterator it;
  for (it = sample_map.begin(); it != sample_map.end(); it++) {
    cout << it->second << " ";
  }
  cout << endl;
}
```

In this example, the map contains pairs of <int, string>, so we create an iterator that matches that format. We use the sample_map.begin() method to point the iterator to where it should start, and indicate that the for loop should stop when we reach the sample_map.end() location—the end of the map.

The iterator provides an it->first function to access the first element in a key-value pair (the key), and then it->second can be used to access the value. So, using the example above, we would print out only the values from our sample map.

## When not to use a C++ map

The map in C++ is a great fit for quickly looking up values by key. However, searching the contents of a map by value requires iterating through an entire map. If you want to be able (dynamic linking), or alternatively include the Boost library inside your executable (static linking).

If you find yourself needing to search a map by value in a relatively simple program, it may be that a map is the wrong object to use. Consider using a C++ vector, queue, stack or other data structure which might end up making the program more straightforward and more efficient.

## Does order matter in C++ maps?

To demonstrate whether order is important in C++ maps, we've slightly modified the above example to insert the element { 4, "four" } before inserting { 3, "three" }:

```
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main() {
  map<int, string> sample_map { { 1, "one"}, { 2, "two" } };
  sample_map.insert({ 4, "four" });
  sample_map[3] = "three";

  map<int, string>::iterator it;
  for (it = sample_map.begin(); it != sample_map.end(); it++) {
    cout << it->second << " ";
  }
  cout << endl;
}
```

When we run the program, the output is still the same as before, even though the order in

This is because maps in C++ keep their elements ordered by key. Also, in C++ a map can't contain duplicate items, so using a map is a way to simultaneously deduplicate and order a set of elements.
If you don't care about the order of elements in a map, consider using an unordered_map element, which is faster at adding and accessing elements than a regular map.

### Can I store a map in a map?

You definitely can! Just make sure that the format of the map reflects your intentions:

```
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main() {
  map<int, map<int, string> > nested_map;
  nested_map[1][1] = "one";
  cout << nested_map[1][1] << endl;
}
```

In this example, the inner map object acts as a value in the outer map.

### Further reading and resources

If you'd like to see the list of all available methods for a map object, check out the map reference. If you're looking for more examples of map usage, consider reading the C++ map tutorial usage guide.

### Conclusion

Back Share

START LEARNING

**STEPHEN WELCH**

Stephen is a Content Developer at Udacity and has built the C++ and Self-Driving Car Engineer Nanodegree programs. He started teaching and coding while completing a Ph.D. in mathematics, and has been passionate about engineering education ever since.

View All Posts by Stephen Welch

**RELATED ARTICLES**

C++ the rule of 5 explained.

After learning about Rvalue references in previous posts, the question of the usefulness of such a construct...

**C++, C++ Move Semantics, C++ Nanodegree, Tech Tutorial, Udacity Instructor Series**

## Exploratory data analysis with Python.

Exploratory data analysis is a critical part of any data analytics or data science process. Before a...

**Exploratory Data Analysis With Python, Tech Tutorial, Udacity Instructor Series**

## Top Five Udacity Blogs You Might Have Missed in Q4 2022

We are always creating blogs to engage our readers in our scholarships, events, and talent transformation efforts....

Back Share

## Exploring C++ Rvalue references.

In the last two posts — Master C++ Copy Semantics and Inside Lvalues & Rvalues in C++...

**C++, C++ Move Semantics, C++ Rvalue References, Tech Tutorial, Udacity Instructor Series**

POPULAR NANODEGREE PROGRAMS +

STUDENT RESOURCES +

UDACITY +

INQUIRIES +

Back

Share