




Course

-  ELEC-A7151

 Course materials

 Your points

Code

-  Code Vault

Course Pages

-  MyCourses

 Teams Channel



This course has already ended.

The latest instance of the course can be found at: [Object oriented programming with C++: 2023 Autumn](#)

« 3 Iterators

Course materials

5 Summary on containers »

ELEC-A7151 / Module 2: Containers / 4 Associative Containers

## 4 Associative Containers¶

### Contents

- 4 Associative Containers
  - 4.1 Defining container and inserting values
  - 4.2 Accessing keys in container
  - 4.3 Iterators

C++ Primer Chapter 11 (Associative containers)

Associative containers access and store data that is associated with a **key**, and can later be accessed using it. As with sequential containers, the container content can be any (built-in or self-defined) C++ type. Type of the key can also be chosen by the programmer, although there are some additional requirements about the properties of the key type.

There are following types of associative containers:

- map**: basic associative array storing key-value pairs, where each key corresponds to at most one stored value, i.e., keys are required to be unique.
- set**: container where key themselves are the values, i.e., key either exists in a set or it does not.
- multimap**: Similar to map, except that each key can be stored multiple times in container.
- multiset**: Similar to set, except that each key/value can be stored multiple times.

These containers are ordered using the key (i.e., the key type needs be able to support ordering). In addition there are unordered containers, such as **unordered\_map**. We will skip these here, but in case you find the useful, feel free to find more information, for example, from [cppreference.com](#). Also, in the following examples we mainly focus on the *map* type, but you can seek information about the other types from [cppreference.com](#) or one of the course books.

To operate with key/value containers such as *map*, we need another type, **pair**. It represents one key/value pair, with two values of the same types as the keys and values of the container. Using the pair type we can insert content to an associative container such as *map*.

### 4.1 Defining container and inserting values¶

The below example illustrates how an associative array is declared, and how the pair type is used to insert a couple elements to the *map* container.

```
1#include <map>
2#include <string>
3#include <iostream>
4
5int main() {
6    // Keys are of string type (that can be ordered). Values are integers.
7    std::map<std::string, int> points;
8
9    // Create one key/value pair compatible with above.
10   std::pair<std::string, int> aGuy = std::pair<std::string, int>("Jaska", 6);
11
12   // Insert it to the container.
13   points.insert(aGuy);
14
15   // 'make_pair' is a helper to create pairs easily.
16   // when using C++17 or later, you can also use std::pair("Liisa", 8)
17   points.insert(std::make_pair("Liisa", 8));
18
19   // Iterate through map, using the automatic type as described above.
20   for (auto it = points.begin(); it != points.end(); it++) {
21       // it->first is the key part of a pair.
22       // it->second is the value part of a pair.
23       std::cout << it->first << " has " << it->second << " points." << std::endl;
24   }
25}
```

Line 7 shows that like all other containers, *map* also uses the template format (angle brackets) to indicate the types of the keys and values in the container. The two types can be different, or the same. Comma is used to separate the two in the angle brackets. The map, *points*, defined here is initially empty.

Line 10 defines a new pair, *aGuy*, that holds a single key/value pair. The types of pair need to match the types in the *map* the pair is going to be used with. Here the type of key is *string*, and type of value is *int*. The pair is initialized with values that correspond to these types, as shown on the line. After we have built a pair, we can insert it to the container, as happens on line 13. Additionally, before C++17 the pair type needs to define the two types inside the angle brackets.

Line 17 inserts another pair into *points*, now using a shorter form, and **make\_pair** (template) function that generates a pair object based on the parameters of the function, and can conveniently be used directly together with, for example, the *insert* function. C++11 provides even easier way to insert key/value pairs, using braces: `points.insert({"Liisa", 8});`. Additionally, after C++17 one can also do it like this: `points.insert(std::pair("Liisa", 8));`. We will get into why the last one does not work before C++17 in module 5.

### 4.2 Accessing keys in container¶

The following continues the previous example by three lines, and shows how an associative container can be accessed, and how the *map* container uses the subscript operation for both reading and adding new data.

```
1#include <map>
2#include <string>
3#include <iostream>
4
5int main() {
6    // Keys are of string type (that can be ordered). Values are integers.
7    std::map<std::string, int> points;
8
9    // Create one key/value pair compatible with above.
10   std::pair<std::string, int> aGuy = std::pair<std::string, int>("Jaska", 6);
11
12   // Insert it to the container.
13   points.insert(aGuy);
14
15   // 'make_pair' is a helper to create pairs easily.
16   // when using C++17 or later, you can also use std::pair("Liisa", 8)
17   points.insert(std::make_pair("Liisa", 8));
18
19   // Copy Jaska's points to a local variable and print to screen.
20   // Note that if key "Jaska" does not exist, a new item is inserted
21   // into the map using the default constructor
22   int jaska_points = points["Jaska"];
23   std::cout << "Points for Jaska: " << jaska_points << std::endl;
24
25   // More secure way of accessing Liisa's points.
26   // If key "Liisa" does not exist, an exception will be thrown.
27   std::cout << "Points for Liisa: " << points.at("Liisa") << std::endl;
28
29   // Causes a new key "Juuso" to be created, with value 4.
30   points["Juuso"] = 4;
31   std::cout << "Points for Juuso: " << points.at("Juuso") << std::endl;
32}
```

Line 22 reads the points for “Jaska”, and stores them in local integer variable. This variable must match the value type given for the container, and the key given inside the subscript brackets must match the key type.

Line 27 is otherwise a normal output stream handling, but it shows that *at* function can also be used to read content matching a given key. The difference between this function and subscript operator is similar as with sequential containers: the *at* function throws an exception if the given key is not found, which does not happen when using the `[]` operator.

Line 30 shows that the subscript operator can also be used to set the values corresponding to the given key in a *map*. If the given key is not yet in the container, it will be added.

### 4.3 Iterators¶

Ordered associative containers such as *map* also support iterators. They are used much the same way as with sequential containers. For iterators to work, it must be possible to place the keys in order. In C++ terms this means that the comparison operator `<` must be defined for the key. For self-defined classes, this operator must be overloaded such that it supports consistent ordering. The previous example shows use of iterators at the end of the main function.

A *map* iterator points to a *pair* that holds one key and value at a time. The below example shows how iterators are used with a map, although for convenience we used *auto* to indicate the iterator type. The first member of the pair (i.e., the key) can be accessed through the member variable *first*, and the second (the value) can be accessed using the member variable *second*. These are members of the *pair* type, not the iterator.

```
1#include <map>
2#include <string>
3#include <iostream>
4
5int main() {
6    // Keys are of string type (that can be ordered). Values are integers.
7    std::map<std::string, int> points;
8
9    // Insert three key/value pairs.
10   points.insert({"Liisa", 6});
11   points.insert({"Jaska", 8});
12   points.insert({"Juuso", 9});
13
14   // Go through the container and print output, this time without 'auto'.
15   for (std::map<std::string, int>::iterator i = points.begin();
16        i != points.end();
17        i++) {
18       std::cout << i->first << " has "
19               << i->second << " points." << std::endl;
20   }
21}
```

When you execute the program, you will notice that the lines are not in the same order than they were added. The order depends on how the comparison is defined for the key type. Here the key type is *string*, where comparisons happen in alphabetical order, which then affects how the iterator walks through the container’s elements.

Iterators are used by some of the *map* member functions: **find(key)** is similar to *at*, except that it returns an iterator pointing to the given key, if it was found in the map. If not, the *find* function returns the *end* iterator, instead of throwing exceptions. Container elements can be deleted using the **erase** function, which is used like the *erase* of sequential containers.

« 3 Iterators

Course materials

5 Summary on containers »