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

« 5 Summary on containers Course materials 6 Round feedback »
ELEC-A7151 / Module 2: Containers / 6 Algorithms

6 Algorithms¶

- **You can download the template for the programming tasks of the module as a** `zip` **file from** `this link`.

Contents

- 6 Algorithms
  - 6.1 Simple read-only algorithm
  - 6.2 Writing algorithms
  - 6.3 Algorithms that extend the container
  - 6.4 Algorithms with Functions
  - 6.5 Programming task

C++ Primer Chapter 10.1

C++ Primer Chapter 10.2 (A first look at the algorithms)

The C++ Library implements a number of *generic algorithms* that can be applied to any type of container, and can be found in the *std* namespace. The algorithms take the start and end iterators as parameters, and then perform some operation on the values between the iterators.

There are different kinds of algorithms:

> - **read-only algorithms**, such as **find** that looks for an element within the container, **count** that counts how many times a particular value can be found in container, or **accumulate** that calculates the sum of the elements in a container.
> - **writing algorithms**, such as **fill** that writes the container elements with given value
> - **reodering algorithms**, such as **sort** that reorders the container elements according to given function

6.1 Simple read-only algorithm¶

The following example shows how *find*, one of the simplest algorithms, works together with the *list* type. The same code would also work for other kinds of sequential containers.

```
1#include <list>
2#include <string>
3#include <iostream>
4#include <algorithm>
5
6int main() {
7    // Set up and initialise a list container consisting of strings.
8    std::list<std::string> starks = { "Bran","Arya","Sansa","Rickon","Robb" };
9
10    // Try to find string "Arya" from container.
11    auto it = std::find(starks.cbegin(), starks.cend(), "Arya");
12
13    // The iterator points to starks.end() if string was not found,
14    // or to first instance of matching element.
15    if (it != starks.end())
16        std::cout << "Found: " << *it << std::endl;
17    else
18        std::cout << "Did not find" << std::endl;
19}
```

The *#include* headers are otherwise familiar, except for **algorithm**, which contains the definitions for the C++ standard library algorithms.

Line 8 initializes a list, *starks*, consisting of strings. Line 11 uses the *find* function on the list. The function does not need to care about the type of the container, as long as it is given the *begin* and *end* iterators that indicate the range from which the given element ("Arya" in this case) is searched. Iterators **cbegin** and **cend** both return **const_iterator**, but otherwise are similar to *begin* and *end*. **const_iterator** is sufficient here, because the algorithm is not going to modify the container content, but the basic non-const iterator would work as well. This example passes the full list to the function, but we could give any other iterator values, e.g. if we only wanted to process a part of the list.

We use the *auto* declaration for the return value of the *find* function, but it will return an iterator to the list, pointing to the first element with a matching string. If the string is not found, the return value is equal to the *end* iterator.

6.2 Writing algorithms¶

Some functions, such as **fill** overwrite the contents of the given range. For example, the following code replaces the first three elements in the *starks* list with another string:

```
1#include <vector>
2#include <iostream>
3
4int main() {
5    // Set up and initialise a list container consisting of strings.
6    std::vector<std::string> starks = { "Bran","Arya","Sansa","Rickon","Robb" };
7
8    // Overwrite the first three elements by a new string.
9    std::fill(starks.begin(), starks.begin() + 3, "Hodor");
10
11    // Output the contents of the container.
12    for (auto i = starks.cbegin(); i != starks.cend(); i++) {
13        std::cout << *i << " ";
14    }
15    std::cout << std::endl;
16}
```

Instead of the previous example, we will now use *vector* instead of *list*. The *fill* function works with either type, but *vector* allows more flexible iterator arithmetics, as done here on line 9, to indicate that we want to process a range from the first element until the third element (begin() + 3 is not included in the fill range). *fill* will replace the elements in this range with "Hodor".

Try and see what program outputs.

6.3 Algorithms that extend the container¶

insert iterator is a special iterator that uses the container's *insert* function whenever the iterator is assigned to.

The function back_inserter returns an insert iterator at the end of a given container. When such iterator is used for writing, the written elements are appended to the end of the container.

Some of the algorithms use an **output iterator** to point to a location where data is written. If a regular iterator is used, the container needs to have enough space for the algorithm to work. If an insert iterator is used, the container can be grown as a result of the algorithm. Here is an example of the copy function together with *back_inserter* that copies contents of an container to a location pointed by iterator by inserting them.

```
1#include <vector>
2#include <list>
3#include <iostream>
4
5int main() {
6    // Define two containers.
7    std::vector<int> vec = {1, 2, 3, 4, 5, 6};
8    std::list<int> ls = {7, 8, 9};
9
10    // Copy 'vec' container, to a location pointed by insert iterator.
11    // back_inserter(ls) points at the end of 'ls' container.
12    std::copy(vec.begin(), vec.end(), std::back_inserter(ls));
13
14    // Output 'ls' container contents. Use auto type and range for.
15    for (auto i : ls) {
16        std::cout << i << " ";
17    }
18    std::cout << std::endl;
19}
```

6.4 Algorithms with Functions¶

Some algorithms can be combined with a function that somehow operates on the container, either by just reading it, or modifying the contents of the container. One such algorithm is **for_each**, which executes a function for each member in the given iterator range. The below example shows one use of *for_each*; outputting each member in the given container range.

```
1#include <iostream>
2#include <vector>
3#include <algorithm>
4
5// Just output the given string.
6void PrintString(const std::string& s) {
7    std::cout << s << " ";
8}
9
10// returns true (1) if 'a' is shorter than 'b', otherwise false (0).
11bool Shorter(const std::string& a, const std::string& b) {
12    return a.size() < b.size();
13}
14
15int main() {
16    // Set up container.
17    std::vector<std::string> starks = { "Bran","Arya","Sansa","Rickon","Robb" };
18
19    // Call 'PrintString' for each member in starks container.
20    // The argument of 'PrintString' must match container element type.
21    std::for_each(starks.begin(), starks.end(), PrintString);
22    std::cout << std::endl;
23
24    // sort the container contents based on default sorting criteria
25    // (alphabetical order with strings)
26    std::sort(starks.begin(), starks.end());
27
28    // output again
29    std::for_each(starks.begin(), starks.end(), PrintString);
30    std::cout << std::endl;
31
32    // sort using a different criteria. Function 'Shorter' defines the order
33    // between the strings.
34    std::sort(starks.begin(), starks.end(), Shorter);
35
36    // print again
37    std::for_each(starks.begin(), starks.end(), PrintString);
38    std::cout << std::endl;
39}
```

Line 21 uses the *for_each* algorithm for the first time. As before, it uses two iterators to indicate the range over which the algorithm is applied. In addition, the third parameter stands for a function, that will be called for each element within the iterator range. In this case, the **PrintString** function just outputs the string element, and as a result, the *for_each* call prints the strings stored in given container. Note that the format of the function is important: it must have exactly one *const* argument that matches the element type in container, and no return value.

Line 26 calls another algorithm, *sort*, that will re-order the contents of the container. The default format with only two arguments (as on this line) is sufficient, if the element type supports comparison (i.e., has the operator `<` defined). For *string* type the default behavior is to compare the data alphabetically, and therefore this variant of *sort* call is possible, and will reorder the container's contents to an alphabetical order.

For data types that do not support comparison by default, or just to override the default comparison method of an type, we can provide alternative function that can be used as as sorting criteria. This is done in the *sort* call on line 34. This variant has a third parameter that indicates the function that implements the sorting method. The *shorter* function compares the lengths of the strings, and returns true if the first string is shorter than the latter. Any other sorting implementation could be used as well, but the function must have exactly two *const* arguments of the same type as the container elements, and it must return a *bool* value. As a result, the container is sorted according to the length of the strings.

6.5 Programming task¶

Points 30 / 30 My submissions 7 Deadline Friday, 8 October 2021, 19:59
To be submitted alone

This course has been archived (Saturday, 17 December 2022, 19:59).

Pokemon

Objective: Practice use of algorithms together with a list container.

Pokemon is an information entity with a name (string) and an identifier. Therefore the **pair** type can be used to store one such element. This exercise operates on lists that consist of (string, size_t) pairs. You should review the function interfaces and respective descriptions there.

You will need to implement the following methods for the *PokemonCollection* class:

- **Add(name, id)**: adds a new Pokemon with given name/id pair at the end of list *pokemon_*.
- **Remove(name, id)**: removes the first Pokemon with matching name and id.
- **Print**: prints the Pokemons. See the example in *pokemon.hpp* for the required print format.
- **SortByName**: sorts the Pokemon collection by their name. If two names are equal, their order is determined by their ids.
- **SortById**: sorts the Pokemon collection by their id. If two ids are equal, their order is determined by their names.

And the following constructor:

- **PokemonCollection(c1, c2)**: merges the contents of the two collections. Duplicate elements must be removed.

The grading weight is 1 for **Add**, **Remove**, **Print**, **SortByName** and **SortById**, and 2 for the merge constructor. Especially for the last three functions it is recommended that you familiarize yourself with the algorithms and functions available for list container (such as **sort** and **unique**).

Note

You are required to use list-specific functions, instead of generic std::sort. See the reference for std::list class.

Warning

You need full points from the Add and Print methods for the rest of the tests to be run. In addition, the sort methods need full points for the constructor to be tested.

Instructions on how to run and test your programs locally are available in Getting Started Module.

pokemon.cpp

Choose File No file chosen

Submit