This course has already ended.
The latest instance of the course can be found at: Object oriented programming with C++: 2023 Autumn

2 Sequential containers¶

Contents

C++ Primer: Chapter 9.1 (Overview of the sequential containers)

C++ standard library includes a number of different kinds of containers for storing data. *Vector* was already shortly introduced, but there are also others. Containers can be further separated into *sequential containers* where data is stored in an ordered way, and *associative containers* where data is associated with a *key*, and possibly stored in unordered way. Containers are implemented in a generic way, such that they can store any (built-in or user-defined) data type. With C++, implementing own data storage classes (such as linked lists) is therefore not usually needed, because the standard library containers is usually sufficient and efficient.

There are the following types of sequential containers (with links to respective reference at *cppreference.com*):

- vector: dynamically sized array, fast insert at the back (otherwise may be slow). Allows random access of any object in the vector.
- deque: double-ended queue, insertion is fast at the beginning or end, otherwise slow. Allows random access of any object.
- list: double-linked list, fast insertion or deletion, but does not allow random access, i.e., the members need to be processed in sequence (remember how linked list works).
- forward_list: single linked list, as above, but list can only be processed to one direction.
- array: fixed size container that does not allow insertion or deletion after initialization. Like builtin C array, but the container class provides safer operations for accessing the content.
- string: sequential container made of characters. Similar properties as vector.

In addition there are containers, such as stack and priority queue. For more information about these containers and their use cases, check the C++ Primer book, or the cpp reference site for the respective container types.

2.1 Setting up a container¶

All sequential containers are used in similar way, and they support the same functions for the most part. Here are examples of different ways of setting up a new container object.

```
1#include <deque>
2#include <list>
3#include <vector>
4#include <string>
5#include "car.hpp"
6
7int main() {
8     std::deque<int> queue;  // double queue of integers, initialized as empty
9     std::list<std::string> first = {"one", "two"};  // two members in a list
10    std::list<std::string> another(first); // copy of first
11    std::vector<Car> cars(5); // Vector with 5 cars (using default constructor)
12    std::vector< std::vector<int> > table;  // Vector of vectors of ints
13}
```

Each type of container is defined in a dedicated standard library header file, therefore the list of *#include* directives is needed in the beginning. By default all containers are initialized as empty, and this is also the case with the parameterless default constructor on line 8, for double-ended queue storing integers. *first* is a linked list containing strings, that initially has two members. The second list (*another*, line 10), is a copy of *first*, with same two members. Line 11 creates a vector of *Car*-type objects (from Module 1) that initially has 5 members. Each member is initialized with the parameterless default constructor in the Car class. Finally, line 12 shows that container member can be another container, like here in the case of vector of vectors.

One should notice how the stored data types are indicated together with the container type in variable declarations, inside angle brackets `<` and `>`. This notation will be seen multiple times in below examples, and are required together with container types, to indicate what the stored data is. When *templates* are discussed later, the meaning of these become clearer.

2.2 Operating on containers¶

The following operations are available with sequential containers:

- **c.size()**: returns the number of items in container *c*. Does not work for *forward_list* type containers.
- **c.max_size()**: returns the maximum possible size for container *c*.
- **c.empty()**: returns *true* (i.e., 1) if container *c* is empty.
- **c.push_back(i)**: Adds element *i* at the end of container *c*. Type of *i* must be compatible with the type given at container definition. *Not available for array or forward_list type containers*.
- **c.push_front(i)**: Adds element *i* to the beginning of container *c*. *Not available for array or vector type containers*
- **c.pop_back()**: Removes the last element in container *c*. *Not available for array or forward_list type containers*.
- **c.pop_front()**: Removes the first element in container *c*. *Not available for array or vector*.

With container types that support random access of data (i.e., reading from anywhere in the container), an individual container element can be accessed using the subscript operator (`[]`), as shown with vector in Module 1. This does not work with *list* or *forward_list*, however, because they are linked lists. For those the elements in the middle of container must be reached first using *iterators*, that will be discussed shortly.

As with C, careless use of subscript operator may accidentally access out of the bounds of the container, and cause unpredictable behavior, possibly crashing the program. However, there is a safer way to access individual members using the **at(index)** function, where *index* is an integer index, just like with the subscript operator. The *at* function is safer: if the index points out of bounds of the container, it throws an *out_of_range exception* instead of causing unpredictable invalid behavior. Handling exceptions will be discussed at a later stage.

The below example illustrates the basic use of a container. Feel free to test and modify it as you wish.

```
#include <deque>
#include <list>
#include <vector>
#include <string>
#include <iostream>

int main() {
    // double queue of integers, initialized as empty
    std::deque<std::string> queue;

    // Add three elements
    queue.push_back("Alvari");
    queue.push_back("Bemari");
    queue.push_back("Cemmari");

    // Take out and delete the first element ("Alvari")
    queue.pop_front();

    // check the status of the queue
    std::cout << "Size now: " << queue.size()
              << " -- is empty: " << (queue.empty() ? "true" : "false")
              << std::endl;
    std::cout << "First item: " << queue[0] << std::endl;

    // one way to walk through the queue
    // more info about 'size_type' below
    for (std::deque<std::string>::size_type i = 0; i < queue.size(); i++) {
        std::cout << i << ": " << queue[i] << std::endl;
    }
}
```

2.3 Type definitions¶

C++ Primer: Chapter 9.2.2 (Container type members)

For each container type there are also some additional type definitions that may be useful, and sometimes needed for correct type management. Like any other names, also type definitions are subject to namespaces.

- **iterator**: type of the basic iterator this container uses. For example: `std::vector::iterator`. Iterators are kind of encapsulated pointers, to one position in container, that can be used to process container contents in sequence (more about them soon).
- **const_iterator**: a constant iterator that does not allow modification of the pointed object. *const_iterator* must be used, if the related container is defined as *const*. For example: `std::queue::const_iterator`
- **size_type**: type of values returned by *size* and *max_size* functions. This is similar to *size_t* in C, and is typically unsigned integer, but for container types there is a separate size_type for each. For example `std::vector::size_type`
- **value_type**: Type of the elements in container. For example `std::vector::value_type`

Each type of container has their own definitions for the above types, and therefore the names are defined under container's namespace. The namespace in use must always be expressed, when using these names.

Using for example *int* in the above example's for loop would not be correct, because it is incompatible with the type returned by *queue.size()*.

2.3.1 auto type¶

C++ Primer: Chapter 2.5.2 (The auto type specifier)

As seen above, the type names in C++ can get long and tedious to type. Fortunately C++ introduces an automatic type, **auto**, that can be used in some places. When *auto* is given as type, the compiler automatically tries to determine the correct type. Sometimes this may result in unwanted guess, as we will see in below example.

Use of the *auto* type has (at least) two benefits:

1. It is shorter to write
2. If the container type is changed at some point during the program's development, the iterator type follows automatically.

To demonstrate *auto* type, below is a modified version of the above example.

```
#include <deque>
#include <list>
#include <vector>
#include <string>
#include <iostream>

int main() {
    // double queue of integers, initialized as empty
    std::deque<std::string> queue;

    // Add three elements
    queue.push_back("Alvari");
    queue.push_back("Bemari");

    // auto determines a correct type for "Cemmari"
    auto somename = "Cemmari";
    queue.push_back(somename);

    // here 'auto' fails:
    // it determines type as 'int', but queue.size() returns unsigned
    // try to fix the resulting warning somehow
    for (auto i = 0; i < queue.size(); i++) {
        std::cout << i << ": " << queue[i] << std::endl;
    }
}
```