# 1. Sequential Containers

Containers can be further separated into sequential containers, and associative containers. Containers are implemented in a generic way (based on templates), such that they can store any (built-in or user-defined) data type. With C++, implementing own data structures (such as linked lists) is therefore not so often needed, because one of the available standard library containers is usually sufficient.

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

* [**vector**](http://www.cplusplus.com/reference/vector/vector/): dynamically sized array, fast insert at the back (otherwise may be slow). Allows random access of any object in the vector.
* [**deque**](http://www.cplusplus.com/reference/deque/deque/): double-ended queue, insertion is fast at the beginning or end, otherwise slow. Allows random access of any object.
* [**list**](http://www.cplusplus.com/reference/list/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**](http://www.cplusplus.com/reference/forward_list/forward_list/): single linked list, as above, but list can only be processed to one direction.
* [**array**](http://www.cplusplus.com/reference/array/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**](http://www.cplusplus.com/reference/string/string/): sequential container made of characters. Similar properties as vector.

In addition there are containers, such as stack and priority queue.

std::deque<int> jono; // double queue of integers, initialized as empty

std::list<std::string> lista = {"one", "two"}; // two members in a list

std::list<std::string> another(lista); // copy of lista

std::vector<Car> cars(5); // Vector with 5 cars (using default constructor)

std::vector< std::vector<int> > table; // Vector of vectors of ints

## 1.2 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* 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.

## 1.3 Type definitions

* iterator: type of the iterator this container uses
* const\_iterator: iterator for types defined as const (i.e., their content is not modifiable). The above does not work for them.
* 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.
* value\_type: Type of the elements in container. This should be the same as given in <..> when declaring a container variable.

std::deque<std::string>::size\_type i = 0

Using for example *int* in the above for loop would not be correct, because it would be incompatible with the type returned by *jono.size()*. It is possible to save typing by using the *auto* type declaration instead of size\_type, but for this example we wanted to make the type explicitly visible.

# 2. Iterators

# Iterator is a pointer to an element within a container. For some types of containers, iterator is the only way to access the container, when direct indexing using [] or at() function is not possible, such as with linked list. Commonly iterators are used to walk through the elements in container, for example in for loop, but they can be used where ever individual elements need to be pointed. For example, some function calls use iterators as parameters.

# Iterator is type named int scope of the container namespace, and each container type has a dedicated iterator type. For example, the basic iterator type for *std::vector* is *std::vector::iterator*, but there are also other iterator types for different kinds of iterators.

You can obtain the iterator corresponding to a container class by using the **begin()** function of the container, which returns an iterator that points to the first element of the container. Alternatively, **end()** returns iterator pointing to the end of the container, more precisely one member after the last member, i.e., trying to access an element at the end() iterator is an error.

Iterators can be compared for equality. If begin() == end(), it means that a container is empty.

The regular iterator type can only be used for modifiable containers. The for loop initializes iterator i using function v.begin(). The loop ends when iterator i becomes equal to v.end().

for (auto i = v.begin();i != v.end(); i++) {

Now the compiler will automatically determine the correct type to use (smart machine!). 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 lifetime, the iterator type follows automatically.

When accessing the date element pointed by iterator, the dereference operator \* is used, as if the iterator was a traditional C pointer (which it is not).

Iterator can be used as a parameter in functions, when individual data elements in a container need to be pointed at. For example, the vector type has function **erase()** that takes an iterator as a parameter. This function causes the pointed item to be deleted from the vector.

Because iterator behaves much like pointer, also iterator arithmetics work similarly as pointer arithmetics, also for larger increments and decrements than one.

Iterators can be used also for string.

The function returns an iterator that points to location after the removed element. Using the return value is important, because **removal may invalidate the previously used iterator**

**Addition and removal of elements in the container may invalidate iterators and references/pointers to the container elements.** For example, sometimes the space allocated for vector needs to be reallocated for addition, which makes the earlier references invalid. Also removal may invalidate iterators. The behavior depends on the container type, and how it internally stores the data..

Another container function that uses iterator as a parameter is **insert\_after(it, e)**, where it is the iterator after which a new element e is inserted. Also this function returns an iterator, that points to the last inserted element.

## 2.1 Range for

int largest = -1000;

for (auto i : v) {

if (i > largest)

largest = i;

}

# Associative container