7.12. When to Use Which Container

The C++ standard library provides different container types with different abilities. The question now is: When do you use which container type? Table 7.57 provides an overview. However, it contains general statements that might not fit in reality. For example, if you manage only a few elements, you can ignore the complexity because short element processing with linear complexity is better than long element processing with logarithmic complexity (in practice, "few" might become very large here).

Table 7.57. Overview of Container Abilities

	Array	Vector	Deque	List	Forward List	Associative Containers	Unordered Containers
Available since	TR1	C++98	C++98	C++98	C++11	C++98	TR1
Typical internal data structure	Static array	Dynamic array	Array of arrays	Doubly linked list	Singly linked list	Binary tree	Hash table
Element type	Value	Value	Value	Value	Value	Set: value Map: key/value	Set: value Map: key/value
Duplicates allowed	Yes	Yes	Yes	Yes	Yes	Only multiset or multimap	Only multiset or multimap
Iterator category	Random access	Random access	Random access	Bidirectional	Forward	Bidirectional (element/key constant)	Forward (element/key constant)
Growing/shrinking	Never	At one end	At both ends	Everywhere	Everywhere	Everywhere	Everywhere
Random access available	Yes	Yes	Yes	No	No	No	Almost
Search/find elements	Slow	Slow	Slow	Very slow	Very slow	Fast	Very fast
Inserting/removing invalidates iterators	—	On reallocation	Always	Never	Never	Never	On rehashing
Inserting/removing invalidates refs/ptrs	—	On reallocation	Always	Never	Never	Never	Never
Allows memory reservation	—	Yes	No	—	—	—	Yes (buckets)
Frees memory for removed elements	—	Only with `shrink_to_fit()`	Sometimes	Always	Always	Always	Sometimes
Transaction safe (success or no effect)	No	Push/pop at the end	Push/pop at the beginning and the end	All insertions and all erasures	All insertions and all erasures	Single-element insertions and all erasures if comparing doesn't throw	Single-element insertions and all erasures if hashing and comparing don't throw

As a supplement to the table, the following rules of thumb might help:

- By default, you should use a vector. It has the simplest internal data structure and provides random access. Thus, data access is convenient and flexible, and data processing is often fast enough.
- If you insert and/or remove elements often at the beginning and the end of a sequence, you should use a deque. You should also use a deque if it is important that the amount of internal memory used by the container shrinks when elements are removed. Also, because a vector usually uses one block of memory for its elements, a deque might be able to contain more elements because it uses several blocks.
- If you insert, remove, and move elements often in the middle of a container, consider using a list. Lists provide special member functions to move elements from one container to another in constant time. Note, however, that because a list provides no random access, you might suffer significant performance penalties on access to elements inside the list if you have only the beginning of the list.

  Like all node-based containers, a list doesn't invalidate iterators that refer to elements, as long as those elements are part of the container. Vectors invalidate all their iterators, pointers, and references whenever they exceed their capacity and part of their iterators, pointers, and references on insertions and deletions. Deques invalidate iterators, pointers, and references when they change their size, respectively.
- If you need a container that handles exceptions so that each operation either succeeds or has no effect, you should use either a list (without calling assignment operations and `sort()` and, if comparing the elements may throw, without calling `merge()`, `remove()`, `remove_if()`, and `unique()`; see Section 7.5.3, page 296) or an associative/unordered container (without calling the multiple-element insert operations and, if copying/assigning the comparison criterion may throw, without calling `swap()` or `erase()`). See Section 6.12.2, page 248, for a general discussion of exception handling in the STL.
- If you often need to search for elements according to a certain criterion, use an unordered set or multiset that hashes according to this criterion. However, hash containers have no ordering, so if you need to rely on element order, you should use a set or a multiset that sorts elements according to the search criterion.
- To process key/value pairs, use an unordered (multi)map or, if the element order matters, a (multi)map.

- If you need an associative array, use an unordered map or, if the element order matters, a map.
- If you need a dictionary, use an unordered multimap or, if the element order matters, a multimap.

A problem that is not easy to solve is how to sort objects according to two different sorting criteria. For example, you might have to keep elements in an order provided by the user while providing search capabilities according to another criterion. As in databases, you need fast access about two or more different criteria. In this case, you could probably use two sets or two maps that share the same objects with different sorting criteria. However, having objects in two collections is a special issue, covered in Section 7.11, page 388.

The automatic sorting of associative containers does not mean that these containers perform better when sorting is needed. This is because an associative container sorts each time a new element gets inserted. An often faster way is to use a sequence container and to sort all elements after they are all inserted, by using one of the several sort algorithms (see Section 11.2.2, page 511).

The following two simple programs sort all strings read from the standard input and print them without duplicates, by using two different containers:

1. Using a **set**:

```
// cont/sortset.cpp

#include <iostream>
#include <string>
#include <algorithm>
#include <iterator>
#include <set>
using namespace std;

int main()
{
    // create a string set
    // - initialized by all words from standard input
    set<string> coll((istream_iterator<string>(cin)),
                     istream_iterator<string>());

    // print all elements
    copy (coll.cbegin(), coll.cend(),
          ostream_iterator<string>(cout, "\n"));
}
```

2. Using a **vector**:

```
// cont/sortvec.cpp

#include <iostream>
#include <string>
#include <algorithm>
#include <iterator>
#include <vector>

using namespace std;

int main()
{
    // create a string vector
    // - initialized by all words from standard input
    vector<string> coll((istream_iterator<string>(cin)),
                        istream_iterator<string>());

    // sort elements
    sort (coll.begin(), coll.end());

    // print all elements ignoring subsequent duplicates
    unique_copy (coll.cbegin(), coll.cend(),
                 ostream_iterator<string>(cout, "\n"));
}
```

When I tried both programs with about 350,000 strings on one system, the vector version was approximately 10% faster. Inserting a call of `reserve()` made the vector version 5% faster. Allowing duplicates — using a `multiset` instead of a `set` and calling `copy()` instead of `unique_copy()`, respectively — changed things dramatically: The vector version was more than 40% faster. However, on another system, the vector versions were up to 50% slower. These measurements are not representative, but they show that it is often worth trying different ways of processing elements.

In practice, predicting which container type is the best is often difficult. The big advantage of the STL is that you can try different versions without much effort. The major work — implementing the different data structures and algorithms — is done. You have only to combine them in a way that is best for you.