**Chapter 15: The Standard Template Library**

Table of Contents

[Introduction to the STL 3](#_Toc49597819)

[Containers 5](#_Toc49597820)

[Sequence Containers 5](#_Toc49597821)

[Associative Containers 7](#_Toc49597822)

[Member Functions 9](#_Toc49597823)

[Container Adapters 10](#_Toc49597824)

[Algorithms 12](#_Toc49597825)

[Iterators 14](#_Toc49597826)

[Potential Problems with the STL 16](#_Toc49597827)

Most computer programs are designed to process data. The data could be of an unending variety, but regardless of what it is, it must be stored in memory and manipulated in similar ways. The way in which data is stored can be encompassed in a general term called a ‘data structure’, while the way in which it is manipulated can be called ‘algorithms’.

C++ classes provide an excellent mechanism for creating a library of data structures. In the past, many vendors and developers offered libraries of container classes to handle the storage and processing of data. Now however, Standard C++ includes its own built-in container class library, called the Standard Template Library (STL). It can be used as a standard approach to storing and processing data.

In this chapter, we will only be covering the very basics of STL.

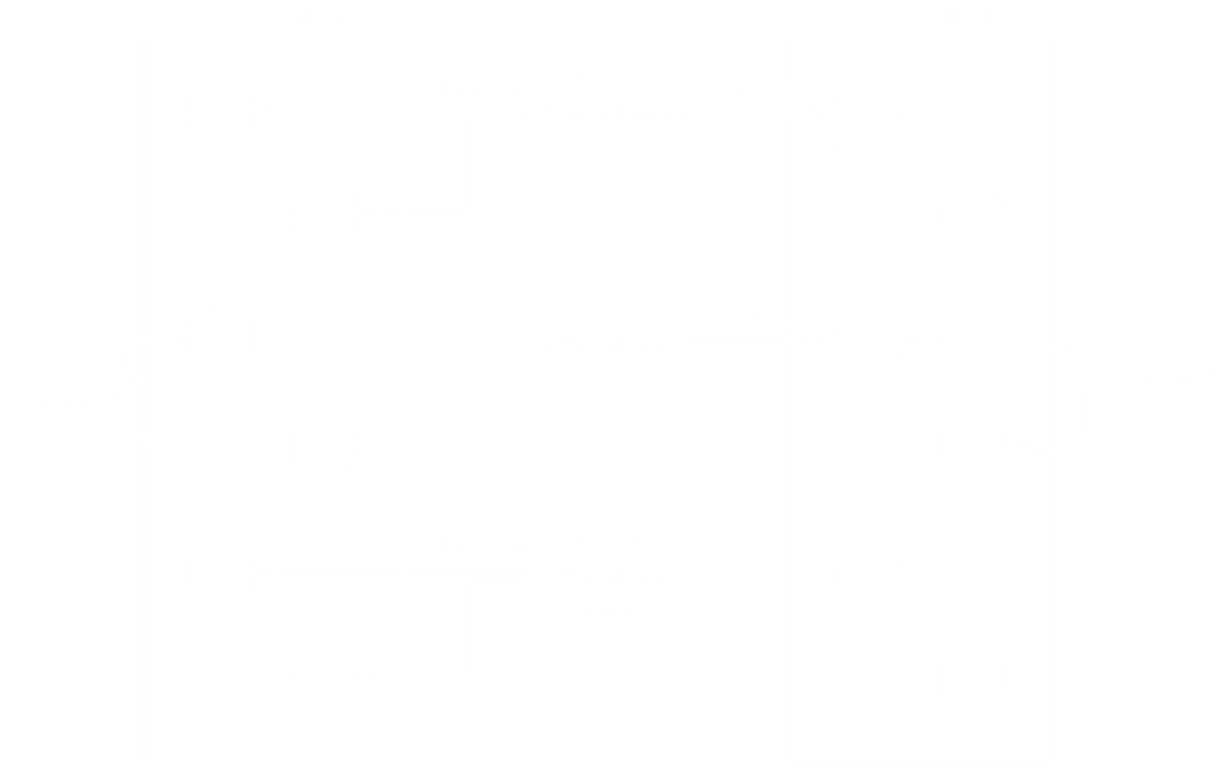
## Introduction to the STL

The STL contains several kinds of entities, the three most important being containers, algorithms and iterators.

A container is a way that stored data is organized in memory. We have explored a few kinds of containers, such as stacks and linked lists. An array is also a type of container, and is so commonly used that it is built directly into C++. The STL includes a host of other useful containers, which are implemented using template classes. This allows them to be easily customized to hold different types of data. Another convenient thing about the STL containers is that, unlike arrays, we do not need to specify any size for the containers. They take care of this themselves.

Algorithms in the STL are procedures that are applied to containers to process their data in various ways. For example, there are algorithms to sort, copy, search and merge data. Algorithms are template functions, but they are not members functions of the container classes. Instead, they are standalone functions. The algorithms are so general that they can be used with ordinary C++ arrays and user-defined containers, not just the STL containers. However, the STL containers do have some member functions for more specific tasks.

Iterators are generalizations of the concept of pointers and are used to point to elements in a container. They can for example, be incremented, just like a pointer can, so that they point to each element in a container in turn. Iterators are a key part of the STL, since they connect algorithms with containers. Algorithms use iterators to reach different members of a container.



## Containers

A container is a way to store data, whether that data is of a built-in type or a class object. The STL includes several basic kinds of containers, three more that are derived from the basic kinds, and also allows us to create our own containers. The reason so many different kinds of containers are made available is efficiency. We could simply use arrays instead of all of the containers we will see, but arrays are awkward and slow in many situations.

The STL containers fall into two categories, sequence and associative. The sequence containers are vector, list and deque. The associative containers are set, multiset and map and multimap. Additionally, there are three containers derived from the sequence containers, namely stack, queue and priority queue.

### Sequence Containers

A sequence container essentially stores elements in a line. Each element is related to the other elements by its position on the line, i.e. each element is preceded by a specific element and followed by a specific element in that order.

An array is an easy example of a sequence container to understand. The problem with an array is that we need to specify its size when we declare it. Most of the time though, we will not be aware of how much data we will need to store in a particular array. The only solution to this is to make the array large enough to hold what we assume will be the maximum amount of data. This leads to either a huge waste of space due to the array not actually being used as heavily as we assumed it would be, or to an error, or even a program crash, due to there not being enough space. The vector container provided in the STL avoids these difficulties.

Another problem with an array is that if we want to insert an element in a specific position, say to keep the array alphabetized, then we will need to move all the elements from that position to the end of the array to create the space needed to insert the new element. This is of course, very time consuming. The STL provides the list container to help us with this. It is based on the idea of a linked list which we know we can insert items into easily simply by rearranging pointers.

The third sequence container, the deque, is like a combination of a stack and a queue. Stacks work in a last-in-first-out fashion, with both input and output taking place at the top of the stack only. Queues work in a first-in first-out fashion, with input taking place at one end of the queue and output taking place at the other. A deque combines these approaches and allows us to insert and remove data from either rend. The word deque is in fact derived from Double Ended Queue. It is a versatile mechanism that is useful in its own right, and can also provide basis for stacks and queues.

|  |  |  |
| --- | --- | --- |
| **Container** | **Characteristic** | **Advantages and Disadvantages** |
| Ordinary C++ Array | Fixed size | + Quick random access (with index).  - Slow to insert or erase in the middle.  - Size cannot be changed at runtime. |
| Vector | Relocation, expandable array | + Quick random access (with index).  - Slow to insert or erase in the middle.  + Quick to insert or erase at the end. |
| List | Doubly linked list | + Quick to insert or delete at any location.  + Quick access to both ends.  - Slow random access. |
| Deque | Like a vector, by can be accessed at either end | + Quick random access (with index).  - Slow to insert or erase in the middle.  + Quick to insert or erase at either the beginning or the end. |

Instantiating an STL container object is easy. The appropriate header file needs to be included, and then we can just use the template format to create the object.

vector<int> aVect; // vector of integers  
list<airTime> departure\_list; // list of airTimes (user-defined class)

C++

### Associative Containers

An associative container uses keys, typically numbers or string, to access data. The containers automatically use the keys to arrange the stored elements in a specific order. Think of a dictionary in which data is accessed by looking up words arranged in alphabetical order. We provide a key value, the container converts this key to the element’s location in memory and gives us the data store at that location. If we know the key, we can access the associated data very quickly.

There are two kinds of associative containers in the STL, sets and maps. They both store data in a structure called a tree, which allows fast searching, insertion and deletion. Sets and maps are very versatile, general data structures that are suitable for many different applications. However, it is inefficient to sort them or perform other operations that require random access.

Sets are simpler and are more commonly used than maps. A set stores a number of items that contain keys. The keys are the attributes used to order the items. For example, we could store objects of a Person class in a set, ordered alphabetically by the name attribute, which acts as the key. As such, we can quickly retrieve a specific object of the Person class by searching for the object with the specified name. If a set stores values of a basic type, such as an integer, the key is the entire item stored. An entire object stored in a set could also be called a key, but we shall call it a key object instead, to emphasize that the attribute used as the key is not necessarily the entire object.

A map stores pairs of objects, a key object and a value object. A map is often used as a container that is somewhat like an array, except that elements are access indices of an arbitrary type instead of being numbers. This means that the key object being stored serves as the index for the associated value.

Maps and sets only allow one key of a given value to be stored. This makes sense, since we expect the keys to be unique. On the other hand, multimaps and multisets allow the same key to be used with multiple values.

|  |  |
| --- | --- |
| **Container** | **Characteristics** |
| set | * Stores only the key objects. * Only one key of each value is allowed. |
| Multiset | * Stores only the key objects. * A single key values can be associated with multiple value objects. |
| Map | * Associates key objects with value objects. * Only one key of each value is allowed. |
| Multimap | * Associates key objects with value objects. * A single key values can be associated with multiple value objects. |

Associative containers are created in the same manner sequential containers are.

set<int> intSet; // a set of integers  
multiset<employee> machinists; // multiset of employees

C++

### Member Functions

Algorithms are the important functions in the STL, carrying out complex operations like sorting and searching. However, containers also need member functions to perform simpler tasks specific to the type of container. The table below gives some member functions whose name and purpose are common to most container classes, even though they may be implemented differently.

|  |  |
| --- | --- |
| **Name** | **Purpose** |
| size() | Returns the number of items in the container |
| empty() | Returns true if the container is empty |
| max\_size() | Returns the size of the largest possible container |
| begin() | Returns an iterator to the start of the container, to allow iterating forwards through the container |
| end() | Returns an iterator to the past-the-end location in the container, used to end forward iteration |
| rbegin() | Returns a reverse iterator to the end of the container, for iterating backwards through the container |
| rend() | Returns a reverse iterator to the beginning of the container, used to end backward iteration |

There are a few more member functions that exist only for certain containers or certain categories of containers.

### Container Adapters

It is possible to create special-purpose containers from the normal ones we have already seen using a construct called container adapters. These special-purpose containers have simpler interfaces than their more general containers. Specialized containers implemented using container adapters in the STL are stacks, queues and priority queues.

A stack restricts access to pushing and popping a data item on and off the top of the stack. A queue adds items to one end and removes them from the other end. In a priority queue, items are added to one end randomly but when removing from the other end, the largest item stored is removed first. This means that the priority queue sorts the data for us.

Stacks, queues and priority queues can be created from different sequence containers, but the deque is often used.

|  |  |  |
| --- | --- | --- |
| **Container** | **Implementation** | **Characteristics** |
| stack | Can be implemented as vector, list or deque | Inserts and removes at one end only |
| queue | Can be implemented as list or deque | Inserts at one end and removes at the other |
| priority queue | Can be implemented as vector or deque | Inserts in random order at one end and removes in sorted order from the other |

These containers are instantiated a little differently. We need to use a template within a template. For example, here is a stack object that holds integers, implemented using a deque class.

stack< deque<int> >aStack;

C++

Notice that a space must be used between the outer pair of angle brackets, since otherwise, the compiler interprets the two repeated angle brackets as the insertion and extraction operators.

## Algorithms

An algorithm is a function that does something to the items in a container. As noted previously, algorithms in the STL are not member functions or even friends of container classes, as they used to be in earlier container libraries. Instead, they are standalone template functions that we can use with built-in C++ arrays and even user-defined classes, provided we implemented the classes with the necessary functions.

|  |  |
| --- | --- |
| **Algorithm** | **Purpose** |
| find | Returns the first element equivalent to a specified value |
| count | Counts the number of elements that have a specified value |
| equal | Compares the contents of two containers and returns true if all corresponding elements are equal |
| search | Looks for a sequence of values in one container that corresponds with the same sequence in another container |
| copy | Copies a sequence of values from one container to another, or to a different location in the same container |
| swap | Exchanges a value in one location with a value in another |
| iter\_swap | Exchanges a sequence of values in one location with a sequence of values in another |
| fill | Copies a value into a sequence of locations |
| sort | Sorts the values in a container according to a specified ordering |
| merge | Combines two sorted ranges of elements to make a larger sorted range |
| accumulate | Returns the sum of the elements in a given range |
| for\_each | Executes a specified function for each element in the container |

Suppose we have an array, arr, that contains integers. We can then use the STL sort() algorithm to sort this array.

Sort(arr, arr + 8);

C++

where arr is the address of the beginning of the array and arr + 8 is the past-the-end address.

## Iterators

Iterators are like pointers, but are not pointers. They are used to access individual data items in a container. Often, they are used to move sequentially from element to elements, also known as iterating through the container. We can increment an iterator using the ++ operator and dereference them using the \* operator to obtain the value of the element they are pointing to. In the STL, an iterator is represented by an object of the iterator class.

Different classes of iterators must be used with different types of containers. There are three major classes of iterators, forward, bidirectional and random access.

A forward iterator can only move forwards through the container, one item at a time. This is accomplished with the ++ operator. It cannot move backwards and it cannot be set to an arbitrary location in the middle of the container.

A bidirectional iterator can move both backwards and forwards using the -- and ++ operators respectively.

A random-access iterator is a bidirectional iterator that can additionally jump to an arbitrary location. We can, for example, tell it to access location 27.

There are also two specialized kinds of iterators, an input iterator and an output iterator. An input iterator can ‘point to’ an input device, such as an input stream or a file, to read sequential data items into a container. An output iterator can ‘point to’ an output device, such as an output stream or file, and write elements from a container to that device.

While the values of forward, bi-directional and random-access iterators can be stored for later use, the values of input and output iterators cannot. This is sensible since the first three point to memory locations but the last two ‘point’ to I/O devices for which the pointer values have no meaning

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Iterator Type** | **Read/Write** | **Can be Saved** | **Direction** | **Access** |
| Random-Access | Both | Yes | Both | Random |
| Bi-Directional | Both | Yes | Both | Linear |
| Forward | Both | Yes | Forward only | Linear |
| Output | Write only | No | Forward only | Linear |
| Input | Read only | No | Forward only | Linear |

## Potential Problems with the STL

The sophistication of the STL’s template classes places a strain on compilers, and not all of them are able to deal with it.

* It is sometimes hard to find errors because the compiler reports them to be deep inside the header file when in reality, they are in the class user’s code. This results in occasionally having to use brute force methods, like commenting out code line by line, to figure out where the error is.
* Pre-compilation of header files, which speeds up compilation dramatically on compilers, may cause problems with the STL.
* The STL may generate spurious compiler warnings, such as “Conversion may lose significant digits.” However, they seem to be harmless and can be ignored.

Minor complains aside, the STL is surprisingly robust and versatile. Errors are generally caught during compilation rather than at runtime, and the different algorithms and containers present a consistent interface, i.e. what works with one container or algorithm will most likely work with another, as long as we use it appropriately.