
7.10. Other STL Containers

The STL is a framework. In addition to the standard container classes, the STL allows you to use other data structures as containers. You can use strings or ordinary arrays as STL containers, or you can write and use special containers that meet special needs. Doing so has the advantage that you can benefit from algorithms, such as sorting or merging, for your own type. Such a framework is a good example of the *Open Closed Principle*: *open* to extension; *closed* to modification.[17]

[17] I first heard of the *Open Closed Principle* from Robert C. Martin, who himself heard it from Bertrand Meyer.

There are three different approaches to making containers "STL-able":

1. **The invasive approach**.[18] You simply provide the interface that an STL container requires. In particular, you need the usual member functions of containers, such as `begin()` and `end()`. This approach is invasive because it requires that a container be written in a certain way.

[18] Instead of *invasive* and *noninvasive*, the terms *intrusive* and *nonintrusive* are sometimes used.

2. **The noninvasive approach**.[18] You write or provide special iterators that are used as an interface between the algorithms and special containers. This approach is noninvasive. All it requires is the ability to step through all the elements of a container, an ability that any container provides in some way.
3. **The wrapper approach**. Combining the two previous approaches, you write a wrapper class that encapsulates any data structure with an STL container-like interface.

This subsection first discusses strings as a standard container, which is an example of the invasive approach. It then covers an important standard container that uses the noninvasive approach: ordinary C-style arrays. However, you can also use the wrapper approach to access data of an ordinary array.

Whoever wants to write an STL container might also support the ability to get parametrized on different allocators. The C++ standard library provides some special functions and classes for programming with allocators and uninitialized memory. See Section 19.3, page 1026, for details.

7.10.1. Strings as STL Containers

The string classes of the C++ standard library (introduced and discussed in Chapter 13) are an example of the invasive approach of writing STL containers. Strings can be considered containers of characters. The characters inside the string build a sequence over which you can iterate to process the individual characters. Thus, the standard string classes provide the container interface of the STL. They provide the `begin()` and `end()` member functions, which return random-access iterators to iterate over a string. The string classes also provide some operations for iterators and iterator adapters. For example, `push_back()` is provided to enable the use of back inserters.

Note that string processing from the STL's point of view is a bit unusual. Normally you process strings as a whole object (you pass, copy, or assign strings). However, when individual character processing is of interest, the ability to use STL algorithms might be helpful. For example, you could read the characters with istream iterators, or you could transform string characters by making them uppercase or lowercase. In addition, by using STL algorithms you can use a special comparison criterion for strings. The standard string interface does not provide that ability.

Section 13.2.14, page 684, discusses the STL aspects of strings in more detail and gives examples.

7.10.2. Ordinary C-Style Arrays as STL Containers

You can use ordinary C-style arrays as STL containers. However, ordinary C-style arrays are not classes, so they don't provide member functions such as `begin()` and `end()`, and you can't define member functions for them. Here, either the noninvasive approach or the wrapper approach must be used.

Using the noninvasive approach is simple. You need only objects that are able to iterate over the elements of an array by using the STL iterator interface. Such iterators already exist: ordinary pointers. An STL design decision was to use the pointer interface for iterators so that you could use ordinary pointers as iterators. This again shows the generic concept of pure abstraction: Anything that *behaves* like an iterator *is* an iterator. In fact, pointers are random-access iterators (see Section 9.2.5, page 438). The following example demonstrates how to use C-style arrays as STL containers since C++11:

Click here to view code image

```
// cont/cstylearray1.cpp

#include <iterator>
#include <vector>
#include <iostream>

int main()
{
    int vals[] = { 33, 67, -4, 13, 5, 2 };
```

```
    // use begin() and end() for ordinary C arrays
    std::vector<int> v(std::begin(vals), std::end(vals));

    // use global begin() and end() for containers:
    std::copy (std::begin(v), std::end(v),
               std::ostream_iterator<int>(std::cout," "));
    std::cout << std::endl;
}
```

Here, we use a helper function defined in `<iterator>` and every container header, which allows using a global `begin()` and `end()` for ordinary C-style arrays. As you can see, for any ordinary C-style array, `vals std::begin()` and `std::end()` yield the corresponding begin and end to use it in the STL framework:

```
int vals[] = { 33, 67, -4, 13, 5, 2 };

std::begin(vals)    // yields vals
std::end(vals)      // yields vals+NumOfElementsIn(vals)
```

These functions are also overloaded, so you can use STL containers or all classes that provide `begin()` and `end()` as member functions:

```
std::vector<int> v;

std::begin(v)       // yields v.begin()
std::end(v)         // yields v.end()
```

The output of the program is as follows:

```
33 67 -4 13 5 2
```

Before C++11, you had to pass the raw pointers to the algorithms because `begin()` and `end()` were not globally provided. For example:

```
// cont/cstylearray1old.cpp

#include <iostream>
#include <algorithm>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    int coll[] = { 5, 6, 2, 4, 1, 3 };

    // square all elements
    transform (coll, coll+6,          // first source
               coll,                  // second source
               coll,                  // destination
               multiplies<int>());    // operation

    // sort beginning with the second element
    sort (coll+1, coll+6);

    // print all elements
    copy (coll, coll+6,
          ostream_iterator<int>(cout," "));
    cout << endl;
}
```

You had to be careful to pass the correct end of the array, as is done here by using `coll+6`. And, as usual, you have to make sure that the end of the range is the position after the last element.

The output of the program is as follows:

```
25 1 4 9 16 36
```

Additional examples for the use of ordinary C-style arrays are in Section 11.7.2, page 579, and in Section 11.10.2, page 620.