The vector<T> Class Template

A QUICK REVIEW OF FUNCTION TEMPLATES

In Section 8.5 we introduced **function templates**, which are patterns for functions from which the compiler can create actual function definitions. A template typically has a **type parameter** that is used as a place holder for a type that will be supplied when the function is called. For example, we considered the following function template:

```
template <typename Item>
void swap(Item & first, Item & second)
{
   Item temporary = first;
   first = second;
   second = temporary;
}
```

When `swap()` is called with

```
swap(int1, int2);
```

the compiler creates an instance of `swap()` in which the type-parameter `Item` is replaced by `int` — the type of the variables whose values are being exchanged. But when `swap()` is called with

```
swap(char1, char2);
```

the compiler creates an instance of `swap()` in which `Item` is replaced by `char`. Function templates thus allow a programmer to create **generic functions**—functions that are *type independent.*

CLASS TEMPLATES

In addition to function templates, C++ also allows **class templates**, which are type-independent patterns from which actual classes can be defined. These are useful for building **generic container classes**—objects that store other objects. In the early 1990s, Alex Stepanov and Meng Lee of Hewlett Packard Laboratories extended C++ with an entire library of class and function templates, which has come to be known as the **Standard Template Library (STL)** and is one of the standard C++ libraries.

One of the simplest containers in STL is the `vector<T>` class template, which can be thought of as a type-independent pattern for a self-contained array class whose capacity may change. Its declaration has the form

```
template <typename T>
class vector
{
   // details of vector omitted...
};
```

where `T` is a parameter for the type of values to be stored in the container. To illustrate its use, consider the following definitions:

```
vector<double> realVector;
vector<string> stringVector;
```

When it processes the first definition, the compiler will create a definition of class vector in which each occurrence of T is replaced by double, and will use this class to construct an object named realVector. When it processes the second definition, the compiler will create a second definition of class vector in which each occurrence of T is replaced by string, and use this class to define an object named stringVector.[1]

Because a class template is a parameterized pattern from which a class can be built and not an actual class itself, its name includes its <T> parameter, as in vector<T>, to distinguish it from an actual class, which is not parameterized.

DEFINING vector<T> OBJECTS

The definition

```
vector<string> empVector;
```

in the program of Figure 10-3 creates an empty vector<string>, meaning a vector that can store string values and whose size and capacity are both zero. As the program demonstrates, having a capacity of zero is not a problem, because a vector<T> object can increase its capacity as necessary during program execution.

Preallocating a vector<T> Object. Sometimes it is useful to *preallocate* the capacity of a vector<T> (see the discussion of push_back() later). This can be done by passing the desired initial capacity as an argument to the object being constructed. For example, if we write

```
vector<string> empVector(10);
```

then empVector will be constructed as a vector of string values whose capacity is 10. Its size will also be 10 since these string values will be initialized (by default) to strings of length 0 that contain no characters:

Unlike a C-style array whose capacity must be known at compile-time, the space for a vector<T> object is allocated during program execution. This allows us to write code like the following that provides *interactive control* over the capacity of a vector<T>:

```
int numberOfEmployees
cin >> numberOfEmployees;
vector<string> empVector(numberOfEmployees);
```

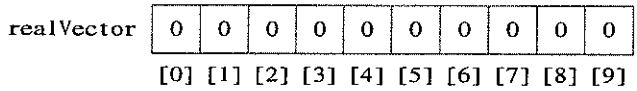
Whatever value the user enters for numberOfEmployees will be the initial capacity (and size) of empVector.

[1] Since class names must be unique, whenever the compiler builds an actual class from a class template, it gives the new class a unique name formed by combining the name of the class and its parameterized type(s), a process known as **name mangling**. For example, vector<double> and vector<string> might be mangled to vector_d and vector_s, respectively.

Preallocating and Initializing a `vector<T>` Object. A third form of a `vector<T>` definition can be used both to preallocate the capacity of a `vector<T>` object and to initialize each element to a specified value. For example, the definition

```
vector<double> realVector(10, 0.0);
```

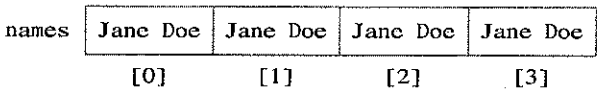
will construct `realVector` as a vector of `double` values, with capacity 10, and containing 10 zeros (making its size also 10):



Similarly, the definition

```
vector<string> names(4, "Jane Doe");
```

will construct `names` as a vector of 4 `string` values, each initialized to `"Jane Doe"`:



The three forms of `vector<T>` definitions can be summarized as follows:

`vector<T>` Definition

Forms:

```
vector<element_type> object_name;

vector<element_type> object_name(initial_capacity);

vector<element_type> object_name(initial_capacity,
                                 initial_value);
```

where:

`element_type` is any known type;

`object_name` is the name of the vector object being defined;

`initial_capacity` is an integer expression; and

`initial_value` is an object of type *`element_type`*.

Purpose:

Define a varying-capacity object capable of storing values of type *`element_type`*.

Form 1 constructs *`object_name`* with capacity and size 0.

Forms 2 and 3 both construct *`object_name`* with capacity *`initial_capacity`* and with size *`initial_capacity`*. Form 2 initializes the elements to default values of type *`element_type`*. Form 3 initializes each element to *`initial_value`*.

`vector<T>` FUNCTION MEMBERS

C++ provides a rich set of operations for `vector<T>`. As we shall see, these operations allow a `vector<T>` to be treated much like an array or `valarray`, but without some of the limitations of C-style arrays and `valarrays`. The following table lists some of these operations:

Operation	Description
`vector<Type> v;`	Construct *v* as a `vector<Type>` of capacity 0
`vector<Type> v(n);`	Construct *v* as a `vector<Type>` of capacity *n*, size *n*, and each element initialized to a default *Type* value
`vector<Type> v(n, initVal);`	Construct *v* as a `vector<Type>` of capacity *n*, size *n*, and each element initialized to *initVal*
`v.capacity()`	Return the number of values *v* can store
`v.size()`	Return the number of values *v* currently contains
`v.empty()`	Return `true` if and only if *v* contains no values (i.e., *v*'s size is 0)
`v.reserve(n);`	Grow *v* so that its capacity is *n* (does not affect *v*'s size)
`v.push_back(value);`	Append *value* at *v*'s end
`v.pop_back();`	Erase *v*'s last element
`v.front()`	Return a reference to *v*'s first element
`v.back()`	Return a reference to *v*'s last element

The Constructors. The first group of operations provide the three main ways to construct a `vector<T>`. We have already described these in some detail.

Checking Size and Capacity. The second group of operations show how much more self-contained `vector<T>` objects are than C-style arrays:

- `v.size()` returns the number of values in *v*. As with C-style arrays and `valarrays`, *the index of the first value of a* `vector<T>` *is always 0*; but unlike arrays and `valarrays`, which provide no way to identify the final values stored in them, *the index of the last value stored in a* `vector<T>` *object v is always* `v.size() - 1`.
- `v.empty()` is a faster alternative to the boolean expression `v.size() == 0`.
- `v.capacity()` returns the current capacity of *v*.
- `v.reserve()` can be used to increase the capacity of *v*, but this is more often done implicitly using `v.push_back()`.

To illustrate the use of these methods, suppose we write

```
vector<int> intVector;
cout << intVector.capacity() << ' '
     << intVector.size() << endl;
```

The output produced will be

```
0 0
```

because `intVector` is an empty container. But if we change the declaration of `intVector` to

```
vector<int> intVector(3);
```

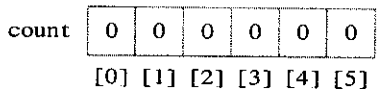
the values

```
3 3
```

will be displayed, because `intVector` has space for 3 values and contains 3 default `int` values (0). The same output will be produced if we use the declaration

```
vector<double> intVector(3, 0);
```

because `realVector` has space for 3 values and contains 3, each of which is 0.0:



Appending and Removing Values. The third group of operations shows how values can be appended to or removed from a `vector<T>`. A statement of the form

```
v.push_back(value);
```

appends *value* to the end of *v*, and increases its size by 1. If necessary, the capacity of *v* is increased to accommodate the new value.

To illustrate, execution of the statements

```
vector<int> intVector;
cout << intVector.capacity() << ' '
     << intVector.size() << endl;
for (int i = 0; i <= 64; i++)
{
   intVector.push_back(i);
   cout << intVector.capacity() << ' '
        << intVector.size() << endl;
}
```

on one machine produced

```
0 0
1 1
2 2
4 3
4 4
8 5
8 6
8 7
8 8
16 9
 .
 .
 .
16 16
32 17
32 18
 .
 .
 .
32 32
64 33
 .
 .
 .
64 64
```

We see that the capacity of `intVector` increased to 1 when the first value was added to it and then doubled each time more space was needed. When the declaration of `intVector` was changed to

```
vector<int> intVector(3, 0);
```

execution of the statements produced

```
3 3
6 4
6 5
6 6
12 7
12 8
 .
 .
 .
12 12
24 13
 .
 .
 .
24 24
48 25
 .
 .
 .
48 48
96 49
 .
 .
 .
96 68
```

Here, we see that the capacity is initially 3, as expected, but when a fourth value is appended to the full `intVector`, its capacity doubles from 3 to 6. Similarly, when the capacity of `intVector` is 6 and we add a seventh value, its capacity doubles again (to 12).

This behavior is a nice compromise between allocating many small blocks of memory (which wastes time) and allocating only a few large blocks of memory (which wastes space). It can be produced by declaring the `vector<T>` as nonempty or by using the function member `reserve()` to set the initial capacity of an empty `vector<T>`. The `reserve()` method can also be used to override the doubling feature of a `vector<T>`'s capacity,

The method *v*`.pop_back()` can be used to remove the last value in *v*. This will decrease the size of *v* by 1, but it does not change its capacity.

Accessing the First and Last Values. The `front()` and `back()` messages can be used to access the first and last values in a `vector<T>`. More precisely, if `realVector` is the vector

realVector	4.3	7.2	5.9
	[0]	[1]	[2]

then `front()` and `back()` can be used to retrieve the first and last values, as in

```
cout << realVector.front() << ' '
     << realVector.back() << endl;
```

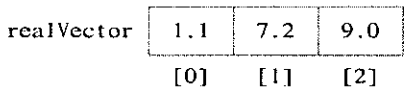
which will display

```
4.3 5.9
```

These methods can also be used to change the first and last values, as in

```
realVector.front() = 1.1;
realVector.back() = 9.0;
```

which will modify `realVector` as follows:



vector<T> OPERATORS

There are four basic **operators** defined in `vector<T>`:

Operator	Description
`v[i]`	Access the element of *v* whose index is `i`
`v1 = v2`	Assign a copy of *v2* to *v1*
`v1 == v2`	Return `true` if and only if *v1* has the same values as *v2*, in the same order
`v1 < v2`	Return `true` if and only if *v1* is lexicographically less than *v2*

The Subscript Operator. The first operator is the familiar subscript operator that provides convenient access to the element with a given index. As we have noted, the index of the first element of a `vector<T>` is always 0, and the expression *vectorName*`.size() - 1` is always the index of the final value in *vectorName*. This allows all of the values stored in a `vector<T>` to be processed using a `for` loop and the subscript operator, as we saw in the program in Figure 10-3:

```
for (int i = 0; i < empVector.size(); i++)   // display
   outStream << empVector[i] << endl;        //  vector
```

The `vector<T>` subscript operation is thus similar to that of `string` objects and C-style arrays.

When Not to Use Subscript. There is one important difference between `vector<T>`s and C-style arrays and `valarrays`. To illustrate this difference, suppose we modified the `read()` function in Figure 10-4 to read values into a `vector<T>` as follows:

```
template <typename T>
void read(istream & in, vector<T> & theVector)
{
   int count = 0;

   for (;;)
   {
      in >> theVector[count]; // LOGIC ERROR:
      if (in.eof()) break;    //  size & capacity not updated!
      count++;
   }
}
```

This does not work correctly, however, because *if the subscript operator is used to append values to a* `vector<T>`, *neither its size nor its capacity is modified. The* `push_back()` *method should always be used to append values to a* `vector<T>`, because it updates the vector's size (and if necessary, its capacity).▽ Figure 10-4 shows one way to write a correct, generic `vector<T>` input function, that uses the end-of-file mark as a sentinel value:

Figure 10-4 `vector<T>` Input from an `istream`

```
/* read() fills a vector<T> with input from a stream.
 *
 * Receives:     type parameter T,
 *               an istream and a vector<T>
 * Input:        a sequence of T values
 * Precondition: operator >> is defined for type T
 * Passes back:  the modified istream and the modified vector<T>
 ****************************************************************/

template <typename T>
void read(istream& in, vector<T>& theVector)
{
   T inputValue;

   for (;;)
   {
      in >> inputValue;
      if ( in.eof() ) break;
      theVector.push_back(inputValue);
   }
}
```

Given this function template, the statements

```
vector<double> realVector;
read(cin, realVector);
```

will cause the compiler to create a definition of `read()` in which each occurrence of `T` has been replaced by `double`, and the call to `read()` will be linked to this definition. If in the same program we write

```
vector<string> stringVector;
read(cin, stringVector);
```

then the compiler will create a second definition of `read()` in which each occurrence of `T` is replaced by `string`, and this second call to `read()` will be linked to this definition.

Because an `ifstream` is a specialized form of istream, this same function template can be used to fill a `vector<T>` from a file by passing an open `ifstream` to that file as an argument to `read()`:

```
ifstream fin("data.txt");
read(fin, realVector);
```

▽ The `push_back()` method also correctly updates the iterator returned by the `end()` function member, while the subscript operator does not. STL algorithms will thus not work properly if subscript is used to append values to a `vector<T>`.

When to Use Subscript. Once a vector<T> contains values, *then and only then* should the subscript operator be used to access (or change) those values. For example, Figure 10-5 presents a generic output function that uses a for loop and the subscript operator to display the values in a vector<T>:

Figure 10-5 vector<T> Output.

```
/* print() writes a vector<T> to a stream.
 *
 * Receives:     type parameter T
 *               an ostream and a vector<T>
 * Output: each T value stored in theVector to ostream out
 * Precondition: operator << is defined for type T
 * Passes back:  the modified ostream
*****************************************************************/

template <typename T>
void print(ostream& out, const vector<T>& theVector)
{
   for (int i = 0; i < theVector.size(); i++)
      out << theVector[i] << ' ';
}
```

Given this function template and the statements

```
vector<double> realVector(5, 1.1);
print(cout, realVector);
```

the compiler will generate a definition of print() in which each occurrence of T is replaced by double. When executed, it will produce the output

```
1.1 1.1 1.1 1.1 1.1
```

Because an ofstream is a specialized form of ostream, this same function template can be used to write a vector<T> to a file by passing an open ofstream to that file as an argument to print():

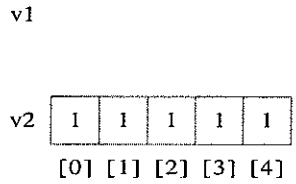
```
ofstream fout("data.txt");
print(fout, realVector);
```

Because these functions are useful in many problems, they should probably be stored in a library (e.g., myVector.h) so that any program can easily reuse them.

The Assignment Operator. The assignment operator (=) is straightforward, behaving exactly as one would expect. For example, the definitions

```
vector<int> v1;
vector<int> v2(5, 1);
```

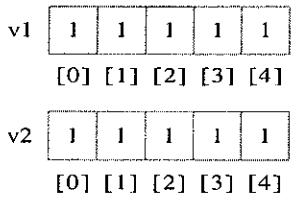
create v1, an empty vector<int>, and v2, a non-empty vector<int>:



A subsequent assignment statement

```
v1 = v2;
```

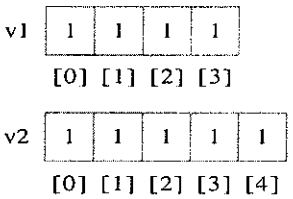
changes `v1` to a copy of `v2`:



The Equality Operator. The equality operator (==) is also straightforward. It compares its operands element by element and returns true if and only if they are identical; that is, their sizes match and their values match. For example, the definitions

```
vector<int> v1(4, 1);
vector<int> v2(5, 1);
```

produce two similar but not identical `vector<int>` objects:



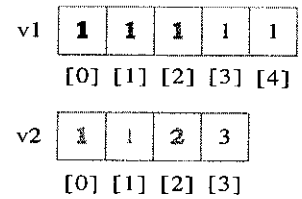
When these are compared using the equality operator,

```
if (v1 == v2)
// ... do something appropriate
```

the expression produces the value `false`, because `v1` and `v2` are not identical.

The Less-Than Operator. The less-than operator (<) is also defined for `vector<T>` objects. It behaves much like the `string` less-than operation, performing an element-by-element comparison until a mismatch (if any) occurs. If the mismatched element in the left operand is less than the corresponding element in the right operand, the operation returns `true`; otherwise it returns `false`. If all the elements of both `vector<T>` objects are compared and no mismatch is found, the operation returns false.

To illustrate, suppose the two `vector<int>` objects `v1` and `v2` have the following values:

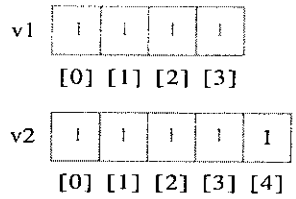


When these are compared using the less-than operator:

```
if (v1 < v2)
// ... do something appropriate
```

the values at index 0 are compared first. Because they are equal, the function moves on and examines the values at index 1. Once again, the values are equal, so it moves on and examines the values at index 2. Now, the value (1) at index 2 in `v1` is less than the value (2) at index 2 in `v2`, the < operation returns `true`.

If the values of `v1` and `v2` are



the values at index 0 are compared. Because they match, the values at index 1 are compared. This continues through the values at index 2 and the function moves on and examines the values at index 3. Because we have reached the end of `v1` but not the end of `v2` and no mismatch has occurred, the operation returns `true`.

`vector<T>` Function Members Involving Iterators. As we have seen, the elements of a `vector<T>` can be accessed using an index and the subscript operator. However, some of the `vector<T>` operations (and those for other STL containers as described in the next section) require a different method of access using objects called iterators Basically, an **iterator** is a class that can "point at" an element of a container by storing its memory address and has built-in operations that can access the value stored there and move from one element to another.

Each STL container provides its own group of iterator types and (at least) two methods that return iterators:

- `begin()`: returns an iterator positioned at the first element in the container
- `end()`: returns an iterator positioned immediately after the last value in the container

The following table includes the `vector<T>` versions of these methods and other important operations that use iterators:

Method	Description
`v.begin()`	Return an iterator positioned at *v*'s first value
`v.end()`	Return an iterator positioned immediately after *v*'s last value
`v.rbegin()`	Return a reverse iterator positioned at *v*'s last value
`v.rend()`	Return a reverse iterator positioned 1 element before *v*'s first value
`v.insert(pos, value);`	Insert *value* into *v* at iterator position *pos*
`v.insert(pos, n, value);`	Insert *n* copies of *value* into *v* at iterator position *pos*
`v.erase(pos);`	Erase the value in *v* at iterator position *pos*
`v.erase(pos1, pos2);`	Erase the values in *v* from iterator positions *pos1* to *pos2*

For `vector<T>`, an iterator declaration has the form

```
vector<double>::iterator iter = initial-value;
```

where the initialization is optional. Three of the important operators on such iterators are:

`iter++`	Moves `iter` forward to the next element of a vector
`iter++`	Moves `iter` backward to the preceding element of a vector
`*iter`	Accesses the value at the position pointed to by `iter`

The following statements shows how iterators could be used in a loop to output a `vector<double>` v:

```
for (vector<double>::iterator iter = v.begin();
                              iter != v.end(); iter++)
   cout << *iter << "  ";
cout << endl;
```

More details about the use of iterators in `vector<T>` can be found on the book's CD and websites.

DECISION: USE A `vector<T>` OR A C-STYLE ARRAY?

We have now seen two different ways to store a sequence of values of the same type: the C-style array and the C++ `vector<T>` class template. (We will not consider `valarrays` here because they can be used only for numeric values.) This raises the question:

When should a C-style array be used and when should a C++ `vector<T>` *be used?*

Whereas the C-style array is a legacy of programming in the early 1970s, the C++ `vector<T>` was designed in the early 1990s, and thus incorporates over 20 years of additional programming wisdom. This design of `vector<T>` (along with the other class templates in STL) gives it some definite advantages over C-style arrays, making it the preferred choice in many situations:

- The capacity of a `vector<T>` can change during execution; the capacity of a C-style array is fixed at compile-time (except as noted later), and cannot be changed without recompiling the program.
- A `vector<T>` is a self-contained object; the C-style array is not. If the same operation can be implemented with either container, the array version will require more parameters.
- A `vector<T>` is a class template, and its function members (augmented with the STL algorithms) provide ready-to-use implementations of many common operations. The C-style array requires that we reinvent the wheel for most operations and `valarray` operations are limited to numeric applications.

But there surely are times when the strengths of an array outweigh the convenience of using a `vector<T>`:

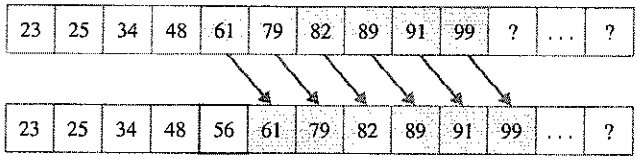
- If one does not need all the operations provided for `vector<T>`s but those for arrays are adequate or nearly so, then a "lean and mean" array that doesn't have all the overhead of `vector<T>` or a class containing an array with a few new operations will perform more efficiently.
- The automatic increases in a `vector<T>`'s capacity can result in considerable waste of memory. For example, if we need to store 1050 elements and start with an empty `vector<T>`, repeated capacity increases will produce a `vector<T>` with 2048 elements, almost twice as many as are needed. Also, each increase in the capacity requires copying the elements from the old container into the new; this can be very time-consuming if the elements are large objects.
- In addition to the arrays we have considered in this chapter, C and C++ also provide run-time allocated arrays (see Chapter 14). These arrays have all the features we have described but also have the additional proprerty that their capacities can be specified during execution.

ARRAY AND `vector<T>` LIMITATIONS

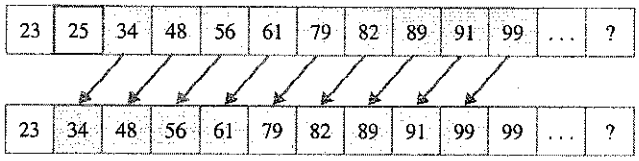
One weakness that arrays, `valarrays`, and the `vector<T>`s all have in common is that inserting and deleting elements at positions other than at the end can be quite time-consuming, especially for large sequences. For example, consider the following ordered sequence of ten integers:

23, 25, 34, 48, 61, 79, 82, 89, 91, 99

If these values are stored in an array, `valarray<int>`, or `vector<int>` and we wish to insert the value 56 into its proper position, then the fifth through the tenth values must be shifted one position to the right to make room for the new value:



Erasing a value from the sequence also requires moving values; for example, to remove the second number, we must shift the third through the eleventh values one position to their left to "close the gap":



If insertions and erasures are restricted to the ends of the sequence, then array and `vector<T>` implementations that do not require moving elements are possible. Two important special cases are stacks and queues. A **stack** is a sequence in which values may be inserted (**pushed**) and removed (**popped**) at only one end, called the **top** of the stack. If elements may be inserted only at one end (the **back**) and removed only at the other (the **front**), the sequence is called a **queue**. STL provides standardized *adaptor* class templates for building such objects out of other containers.

In summary, arrays and `vector<T>` objects work well for storing sequences in which insertions and deletions are infrequent or are restricted to the ends of the list. **Dynamic** sequences whose sizes may vary greatly during processing and those in which items are frequently inserted and/or deleted anywhere in the sequence are better stored in *linked lists*,