7.3. Vectors

A vector models a dynamic array. Thus, a vector is an abstraction that manages its elements with a dynamic C-style array (Figure 7.2). However, the standard does not specify that the implementation uses a dynamic array. Rather, this follows from the constraints and specification of the complexity of its operation.

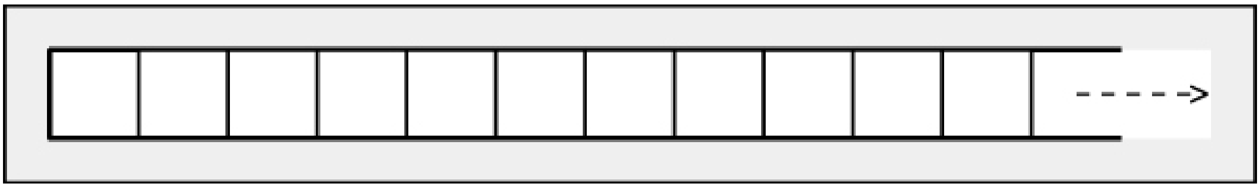

Figure 7.2. Structure of a Vector

To use a vector, you must include the header file `<vector>` :

```
#include <vector>
```

There, the type is defined as a class template inside namespace `std` :

```
namespace std {
    template <typename T,
              typename Allocator = allocator<T> >
    class  vector;
}
```

The elements of a vector may have any type `T` . The optional second template parameter defines the memory model (see Chapter 19). The default memory model is the model `allocator` , which is provided by the C++ standard library.

7.3.1. Abilities of Vectors

A vector copies its elements into its internal dynamic array. The elements always have a certain order. Thus, a vector is a kind of *ordered collection*. A vector provides *random access*. Thus, you can access every element directly in constant time, provided that you know its position. The iterators are random-access iterators, so you can use any algorithm of the STL.

Vectors provide good performance if you append or delete elements at the end. If you insert or delete in the middle or at the beginning, performance gets worse. This is because every element behind has to be moved to another position. In fact, the assignment operator would be called for every following element.

Size and Capacity

Part of the way in which vectors give good performance is by allocating more memory than they need to contain all their elements. To use vectors effectively and correctly, you should understand how size and capacity cooperate in a vector.

Vectors provide the usual size operations `size()` , `empty()` , and `max_size()` (see Section 7.1.2, page 254). An additional “size” operation is the `capacity()` function, which returns the number of elements a vector could contain in its actual memory. If you exceed the `capacity()` , the vector has to reallocate its internal memory.

The capacity of a vector is important for two reasons:

1. Reallocation invalidates all references, pointers, and iterators for elements of the vector.
2. Reallocation takes time.

Thus, if a program manages pointers, references, or iterators into a vector, or if speed is a goal, it is important to take the capacity into account.

To avoid reallocation, you can use `reserve()` to ensure a certain capacity before you really need it. In this way, you can ensure that references remain valid as long as the capacity is not exceeded:

```
std::vector<int> v;    // create an empty vector
v.reserve(80);         // reserve memory for 80 elements
```

Another way to avoid reallocation is to initialize a vector with enough elements by passing additional arguments to the constructor. For example, if you pass a numeric value as parameter, it is taken as the starting size of the vector:

Click here to view code image

```
std::vector<T> v(5);    // creates a vector and initializes it with five values
                        // (calls five times the default constructor of type T)
```

Of course, the type of the elements must provide a default constructor for this ability. For fundamental types, zero initialization (see Section 3.2.1, page 37) is guaranteed. But note that for complex types, even if a default constructor is provided, the initialization takes time. If the only reason for initialization is to reserve memory, you should use `reserve()`.

The concept of capacity for vectors is similar to that for strings (see Section 13.2.5, page 669), with one big difference: Unlike for strings, it is not possible to call `reserve()` for vectors to shrink the capacity. Calling `reserve()` with an argument that is less than the current capacity is a no-op. Furthermore, how to reach an optimal performance regarding speed and memory use is implementation defined. Thus, implementations might increase capacity in larger steps. In fact, to avoid internal fragmentation, many implementations allocate a whole block of memory (such as 2K) the first time you insert anything if you don't call `reserve()` first yourself. This can waste a lot of memory if you have many vectors with only a few small elements.

Because the capacity of vectors never shrinks, it is guaranteed that references, pointers, and iterators remain valid even when elements are deleted, provided that they refer to a position before the manipulated elements. However, insertions invalidate all references, pointers, and iterators when the capacity gets exceeded.

C++11 introduced a new member function for vectors: a nonbinding request to shrink the capacity to fit the current number of elements:

```
v.shrink_to_fit();    // request to shrink memory (since C++11)
```

This request is nonbinding to allow latitude for implementation-specific optimizations. Thus, you cannot expect that afterward `v.capacity()==v.size()` yields `true`.

Before C++11, you could only indirectly shrink the capacity: Swapping the contents with another vector swaps the capacity. Thus, the following function shrinks the capacity while preserving the elements:

```
template <typename T>
void shrinkCapacity(std::vector<T>& v)
{
    std::vector<T> tmp(v);    // copy elements into a new vector
    v.swap(tmp);              // swap internal vector data
}
```

You could even shrink the capacity without calling this function by calling the following statement:[5]

[5] You (or your compiler) might consider this statement as being incorrect because it calls a nonconstant member function for a temporary value. However, standard C++ allows you to call a nonconstant member function for temporary values.

```
// shrink capacity of vector v for type T
std::vector<T>(v).swap(v);
```

However, note that after `swap()`, all references, pointers, and iterators swap their containers. They still refer to the elements to which they referred on entry. Thus, `shrinkCapacity()` invalidates all references, pointers, and iterators. The same is true for `shrink_to_fit()`.

7.3.2. Vector Operations

Create, Copy, and Destroy

Table 7.9 lists the constructors and destructors for vectors. You can create vectors with and without elements for initialization. If you pass only the size, the elements are created with their default constructor. Note that an explicit call of the default constructor also initializes fundamental types, such as `int`, with zero (see Section 3.2.1, page 37). See Section 7.1.2, page 254, for some remarks about possible initialization sources.

Table 7.9. Constructors and Destructor of Vectors

Operation	Effect
`vector<Elem> c`	Default constructor; creates an empty vector without any elements
`vector<Elem> c(`*c2*`)`	Copy constructor; creates a new vector as a copy of *c2* (all elements are copied)
`vector<Elem> c = `*c2*	Copy constructor; creates a new vector as a copy of *c2* (all elements are copied)
`vector<Elem> c(`*rv*`)`	Move constructor; creates a new vector, taking the contents of the rvalue *rv* (since C++11)
`vector<Elem> c = `*rv*	Move constructor; creates a new vector, taking the contents of the rvalue *rv* (since C++11)
`vector<Elem> c(`*n*`)`	Creates a vector with *n* elements created by the default constructor
`vector<Elem> c(`*n*`,`*elem*`)`	Creates a vector initialized with *n* copies of element *elem*
`vector<Elem> c(`*beg*`,`*end*`)`	Creates a vector initialized with the elements of the range [*beg*,*end*)
`vector<Elem> c(`*initlist*`)`	Creates a vector initialized with the elements of initializer list *initlist* (since C++11)
`vector<Elem> c = `*initlist*	Creates a vector initialized with the elements of initializer list *initlist* (since C++11)
`c.~vector()`	Destroys all elements and frees the memory

Nonmodifying Operations

Table 7.10 lists all nonmodifying operations of vectors.[6] See additional remarks in Section 7.1.2, page 254, and Section 7.3.1, page 270.

[6] `reserve()` and `shrink_to_fit()` manipulate the vector because they invalidate references, pointers, and iterators to elements. However, they are mentioned here because they do not manipulate the logical contents of the container.

Table 7.10. Nonmodifying Operations of Vectors

Operation	Effect
`c.empty()`	Returns whether the container is empty (equivalent to `size()==0` but might be faster)
`c.size()`	Returns the current number of elements
`c.max_size()`	Returns the maximum number of elements possible
`c.capacity()`	Returns the maximum possible number of elements without reallocation
`c.reserve(`*num*`)`	Enlarges capacity, if not enough yet[6]
`c.shrink_to_fit()`	Request to reduce capacity to fit number of elements (since C++11)[6]
c1 `==` *c2*	Returns whether *c1* is equal to *c2* (calls `==` for the elements)
c1 `!=` *c2*	Returns whether *c1* is not equal to *c2* (equivalent to `!(`*c1*`==`*c2*`)`)
c1 `<` *c2*	Returns whether *c1* is less than *c2*
c1 `>` *c2*	Returns whether *c1* is greater than *c2* (equivalent to *c2*`<`*c1*)
c1 `<=` *c2*	Returns whether *c1* is less than or equal to *c2* (equivalent to `!(`*c2*`<`*c1*`)`)
c1 `>=` *c2*	Returns whether *c1* is greater than or equal to *c2* (equivalent to `!(`*c1*`<`*c2*`)`)

Assignments

Table 7.11 lists the ways to assign new elements while removing all existing elements. The set of `assign()` functions matches the set of constructors. You can use different sources for assignments (containers, arrays, standard input) similar to those described for constructors (see Section 7.1.2, page 254). All assignment operations call the default constructor, copy constructor, assignment operator, and/or destructor of the element type, depending on how the number of elements changes. For example:

```
std::list<Elem> l;
std::vector<Elem> coll;
```

```
// make coll be a copy of the contents of l
coll.assign(l.begin(),l.end());
```

Table 7.11. Assignment Operations of Vectors

Operation	Effect
`c = c2`	Assigns all elements of *c2* to `c`
`c = rv`	Move assigns all elements of the rvalue *rv* to `c` (since C++11)
`c = initlist`	Assigns all elements of the initializer list *initlist* to `c` (since C++11)
`c.assign(n,elem)`	Assigns *n* copies of element *elem*
`c.assign(beg,end)`	Assigns the elements of the range [*beg*,*end*)
`c.assign(initlist)`	Assigns all the elements of the initializer list *initlist*
`c1.swap(c2)`	Swaps the data of *c1* and *c2*
`swap(c1,c2)`	Swaps the data of *c1* and *c2*

Element Access

To access all elements of a vector, you must use range-based `for` loops (see Section 3.1.4, page 17), specific operations, or iterators. Table 7.12 shows all vector operations for direct element access. As usual in C and C++, the first element has index `0`, and the last element has index `size()-1`. Thus, the *n*th element has index *n* `-1`. For nonconstant vectors, these operations return a reference to the element. Thus, you could modify an element by using one of these operations, provided it is not forbidden for other reasons.

Table 7.12. Direct Element Access of Vectors

Operation	Effect
`c[idx]`	Returns the element with index *idx* (*no* range checking)
`c.at(idx)`	Returns the element with index *idx* (throws range-error exception if *idx* is out of range)
`c.front()`	Returns the first element (*no* check whether a first element exists)
`c.back()`	Returns the last element (*no* check whether a last element exists)

The most important issue for the caller is whether these operations perform range checking. Only `at()` performs range checking. If the index is out of range, `at()` throws an `out_of_range` exception (see Section 4.3, page 41). All other functions do *not* check. A range error results in undefined behavior. Calling operator `[]`, `front()`, and `back()` for an empty container always results in undefined behavior:

Click here to view code image

```
std::vector<Elem> coll;          // empty!

coll[5] = elem;                  // RUNTIME ERROR    undefined behavior
std::cout << coll.front();       // RUNTIME ERROR    undefined behavior
```

So, you must ensure that the index for operator `[]` is valid and that the container is not empty when either `front()` or `back()` is called:

```
std::vector<Elem> coll;        // empty!

if (coll.size() > 5) {
    coll[5] = elem;            // OK
}
if (!coll.empty()) {
    cout << coll.front();      // OK
}
coll.at(5) = elem;             // throws out_of_range exception
```

Note that this code is OK only in single-threaded environments. In multithreaded contexts, you need synchronization mechanisms to ensure that `coll` is not modified between the check for its size and the access to the element (see Section 18.4.3, page 984, for details).

Iterator Functions

Vectors provide the usual operations to get iterators (Table 7.13). Vector iterators are random-access iterators (see Section 9.2, page 433, for a discussion of iterator categories). Thus, in principle you could use all algorithms of the STL.

Table 7.13. Iterator Operations of Vectors

Operation	Effect
`c.begin()`	Returns a random-access iterator for the first element
`c.end()`	Returns a random-access iterator for the position after the last element
`c.cbegin()`	Returns a constant random-access iterator for the first element (since C++11)
`c.cend()`	Returns a constant random-access iterator for the position after the last element (since C++11)
`c.rbegin()`	Returns a reverse iterator for the first element of a reverse iteration
`c.rend()`	Returns a reverse iterator for the position after the last element of a reverse iteration
`c.crbegin()`	Returns a constant reverse iterator for the first element of a reverse iteration (since C++11)
`c.crend()`	Returns a constant reverse iterator for the position after the last element of a reverse iteration (since C++11)

The exact type of these iterators is implementation defined. For vectors, however, the iterators returned by `begin()`, `cbegin()`, `end()`, and `cend()` are often ordinary pointers, which is fine because vectors usually use a C-style array for the elements and ordinary pointers provide the interface of random-access iterators. However, you can't count on the fact that the iterators are ordinary pointers. For example, if a safe version of the STL that checks range errors and other potential problems is used, the iterator type is usually an auxiliary class. See Section 9.2.6, page 440, for a nasty difference between iterators implemented as pointers and iterators implemented as classes.

Iterators remain valid until an element with a smaller index gets inserted or removed or until reallocation occurs and capacity changes (see Section 7.3.1, page 270).

Inserting and Removing Elements

Table 7.14 shows the operations provided for vectors to insert or to remove elements. As usual when using the STL, you must ensure that the arguments are valid. Iterators must refer to valid positions, and the beginning of a range must have a position that is not behind the end.

Table 7.14. Insert and Remove Operations of Vectors

Operation	Effect
`c.push_back(`*elem*`)`	Appends a copy of *elem* at the end
`c.pop_back()`	Removes the last element (does not return it)
`c.insert(`*pos*`,`*elem*`)`	Inserts a copy of *elem* before iterator position *pos* and returns the position of the new element
`c.insert(`*pos*`,`*n*`,`*elem*`)`	Inserts *n* copies of *elem* before iterator position *pos* and returns the position of the first new element (or *pos* if there is no new element)
`c.insert(`*pos*`,`*beg*`,`*end*`)`	Inserts a copy of all elements of the range [*beg*,*end*) before iterator position *pos* and returns the position of the first new element (or `pos` if there is no new element)
`c.insert(`*pos*`,`*initlist*`)`	Inserts a copy of all elements of the initializer list *initlist* before iterator position *pos* and returns the position of the first new element (or *pos* if there is no new element; since C++11)
`c.emplace(`*pos*`,`*args...*`)`	Inserts a new element initialized with *args* before iterator position *pos* and returns the position of the new element (since C++11)
`c.emplace_back(`*args...*`)`	Appends a new element initialized with *args* at the end (returns nothing; since C++11)
`c.erase(`*pos*`)`	Removes the element at iterator position *pos* and returns the position of the next element
`c.erase(`*beg*`,`*end*`)`	Removes all elements of the range [*beg*,*end*) and returns the position of the next element
`c.resize(`*num*`)`	Changes the number of elements to *num* (if `size()` grows new elements are created by their default constructor)
`c.resize(`*num*`,`*elem*`)`	Changes the number of elements to *num* (if `size()` grows new elements are copies of *elem*)
`c.clear()`	Removes all elements (empties the container)

As usual, it is up to the programmer to ensure that the container is not empty when `pop_back()` is called. For example:

Click here to view code image

```
std::vector<Elem> coll;         // empty!
coll.pop_back();                // RUNTIME ERROR     undefined behavior
if (!coll.empty()) {
    coll.pop_back();            // OK
}
```

However, note that in a multithreaded context you have to ensure that `coll` doesn't get modified between the check for being empty and `pop_back()` (see Section 18.4.3, page 984).

Regarding performance, you should consider that inserting and removing happens faster when

- Elements are inserted or removed at the end.
- The capacity is large enough on entry.
- Multiple elements are inserted by a single call rather than by multiple calls.

Inserting or removing elements invalidates references, pointers, and iterators that refer to the following elements. An insertion that causes reallocation invalidates all references, iterators, and pointers.

Vectors provide no operation to remove elements directly that have a certain value. You must use an algorithm to do this. For example, the following statement removes all elements that have the value `val`:

```
std::vector<Elem> coll;
...
// remove all elements with value val
coll.erase(remove(coll.begin(),coll.end(),
                  val),
           coll.end());
```

This statement is explained in Section 6.7.1, page 218.

To remove only the first element that has a certain value, you must use the following statements:

```
std::vector<Elem> coll;
...
// remove first element with value val
std::vector<Elem>::iterator pos;
pos = find(coll.begin(),coll.end(),
           val);
if (pos != coll.end()) {
    coll.erase(pos);
}
```

7.3.3. Using Vectors as C-Style Arrays

As for class `array<>`, the C++ standard library guarantees that the elements of a vector are in contiguous memory. Thus, you can expect that for any valid index `i` in vector `v`, the following yields `true`:

```
&v[i] == &v[0] + i
```

This guarantee has some important consequences. It simply means that you can use a vector in all cases in which you could use a dynamic array. For example, you can use a vector to hold data of ordinary C-strings of type `char*` or `const char*`:

Click here to view code image

```
std::vector<char> v;            // create vector as dynamic array of chars

v.resize(41);                   // make room for 41 characters (including '\0')
strcpy(&v[0],"hello, world");   // copy a C-string into the vector
printf("%s\n", &v[0]);          // print contents of the vector as C-string
```

Note, however, that since C++11, you don't have to use the expression `&a[0]` to get direct access to the elements in the vector, because the member function `data()` is provided for this purpose:

Click here to view code image

```
std::vector<char> v;            // create vector as dynamic array of chars

strcpy(v.data(),"hello, world");   // copy a C-string into the array
printf("%s\n", v.data());          // print contents of the array as C-string
```

Of course, you have to be careful when you use a vector in this way (just as you always have to be careful when using ordinary C-style arrays and pointers). For example, you have to ensure that the size of the vector is big enough to copy some data into it and that you have an `'\0'` element at the end if you use the contents as a C-string. However, this example shows that whenever you need an array of type `T` for any reason, such as for an existing C library, you can use a `vector<T>` and pass the address of the first element.

Note that you must not pass an iterator as the address of the first element. Iterators of vectors have an implementation-specific type, which may be totally different from an ordinary pointer:

Click here to view code image

```
printf("%s\n", v.begin());      // ERROR (might work, but not portable)
printf("%s\n", v.data());       // OK (since C++11)
printf("%s\n", &v[0]);          // OK, but data() is better
```

7.3.4. Exception Handling

Vectors provide only minimal support for logical error checking. The only member function for which the standard requires that it may throw an exception is `at()`, which is the safe version of the subscript operator (see Section 7.3.2, page 274). In addition, the standard requires that only the usual standard exceptions may occur, such as `bad_alloc` for a lack of memory or exceptions of user-defined operations.

If functions called by a vector (functions for the element type or functions that are user-supplied) throw exceptions, the C++ standard library provides the following guarantees:

1. `push_back()` and `emplace_back()` either succeed or have no effect. Note however, that to ensure this behavior, copy constructors instead of move constructors are used for reallocation if move constructors do not guarantee not to throw. Thus, providing a nothrow/noexcept specification for the move operations of the elements will lead to better performance.
2. `insert()`, `emplace()`, and `push_back()` either succeed or have no effect, provided that the copy/move operations (constructors and assignment operators) of the elements do not throw.
3. `pop_back()` does not throw any exceptions.
4. `erase()` does not throw if the copy/move operations (constructors and assignment operators) of the elements do not throw.
5. `swap()` and `clear()` do not throw.
6. If elements are used that never throw exceptions on copy/move operations (constructors and assignment operators), every operation

is either successful or has no effect. Such elements might be “plain old data” (POD). POD describes types that use no special C++ feature. For example, every ordinary C structure is POD.

All these guarantees are based on the requirements that destructors don’t throw. See Section 6.12.2, page 248, for a general discussion of exception handling in the STL.

7.3.5. Examples of Using Vectors

The following example shows a simple use of vectors:

Click here to view code image

```
// cont/vector1.cpp

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

int main()
{
    // create empty vector for strings
    vector<string> sentence;

    // reserve memory for five elements to avoid reallocation
    sentence.reserve(5);

    // append some elements
    sentence.push_back("Hello,");
    sentence.insert(sentence.end(),{"how","are","you","?"});

    // print elements separated with spaces
    copy (sentence.cbegin(), sentence.cend(),
          ostream_iterator<string>(cout," "));
    cout << endl;

    // print "technical data"
    cout << "  max_size(): " << sentence.max_size() << endl;
    cout << "  size():     " << sentence.size()     << endl;
    cout << "  capacity(): " << sentence.capacity() << endl;

    // swap second and fourth element
    swap (sentence[1], sentence[3]);

    // insert element "always" before element "?"
    sentence.insert (find(sentence.begin(),sentence.end(),"?"),
                     "always");

    // assign "!" to the last element
    sentence.back() = "!";

    // print elements separated with spaces
    copy (sentence.cbegin(), sentence.cend(),
          ostream_iterator<string>(cout," "));
    cout << endl;

    // print some "technical data" again
    cout << "  size():     " << sentence.size()     << endl;
    cout << "  capacity(): " << sentence.capacity() << endl;

    // delete last two elements
    sentence.pop_back();
    sentence.pop_back();
    // shrink capacity (since C++11)
    sentence.shrink_to_fit();

    // print some "technical data" again
    cout << "  size():     " << sentence.size()     << endl;
    cout << "  capacity(): " << sentence.capacity() << endl;
}
```

The output of the program might look like this:

```
Hello, how are you ?
  max_size(): 1073741823
  size():     5
  capacity(): 5
```

```
Hello, you are how always !
  size():      6
  capacity(): 10
  size():      4
  capacity(): 4
```

Note my use of the word *might*. The values of `max_size()` and `capacity()` are unspecified and might vary from platform to platform. Here, for example, you can see that the implementation seems to double the capacity if the capacity no longer fits and not necessarily shrinks if there is a request to do so.

7.3.6. Class `vector<bool>`

For Boolean elements, the C++ standard library provides a specialization of `vector<>`. The goal is to have a version that is optimized to use less size than a usual implementation of `vector<>` for type `bool`. Such a usual implementation would reserve at least 1 byte for each element. The `vector<bool>` specialization usually uses internally only 1 bit for an element, so it is typically eight times smaller. But such an optimization also has a snag: In C++, the smallest addressable value must have a size of at least 1 byte. Thus, such a specialization of a vector needs special handling for references and iterators.

As a result, a `vector<bool>` does not meet all requirements of other vectors. For example, a `vector<bool>::reference` is not a true lvalue and `vector<bool>::iterator` is not a random-access iterator. Therefore, template code might work for vectors of any type except `bool`. In addition, `vector<bool>` might perform worse than normal implementations, because element operations have to be transformed into bit operations. However, how `vector<bool>` is implemented is implementation specific. Thus, the performance (speed and memory) might differ.

Note that class `vector<bool>` is more than a specialization of `vector<>` for `bool`. It also provides some special bit operations. You can handle bits or flags in a more convenient way.

`vector<bool>` has a dynamic size, so you can consider it a bitfield with dynamic size. Thus, you can add and remove bits. If you need a bitfield with static size, you should use `bitset` rather than a `vector<bool>`. Class `bitset` is covered in Section 12.5, page 650.

The additional operations of `vector<bool>` are shown in Table 7.15.

Table 7.15. Special Operations of `vector<bool>`

Operation	Effect
`c.flip()`	Negates all Boolean elements (complement of all bits)
`c[`*idx*`].flip()`	Negates the Boolean element with index *idx* (complement of a single bit)
`c[`*idx*`] = `*val*	Assigns *val* to the Boolean element with index *idx* (assignment to a single bit)
`c[`*idx1*`] = c[`*idx2*`]`	Assigns the value of the element with index *idx2* to the element with index *idx1*

The operation `flip()`, which processes the complement, can be called for all bits and a single bit of the vector. The latter is remarkable because you might expect the operator `[]` to return a `bool` and that calling `flip()` for such a fundamental type is not possible. Here, the class `vector<bool>` uses a common trick, called a *proxy*:[7] For `vector<bool>`, the return type of the subscript operator (and other operators that return an element) is an auxiliary class. If you need the return value to be `bool`, an automatic type conversion is used. For other operations, the member functions are provided. The relevant part of the declaration of `vector<bool>` looks like this:

[7] A proxy allows you to keep control where usually no control is provided. This is often used to get more security. In this case, the proxy maintains control to allow certain operations, although, in principle, the return value behaves as `bool`.

Click here to view code image

```
namespace std {
  template <typename Allocator> class vector<bool,Allocator> {
    public:
      // auxiliary proxy type for element modifications:
      class reference {
        ...
        public:
          reference& operator= (const bool) noexcept;   // assignments
          reference& operator= (const reference&) noexcept;
          operator bool() const noexcept;   // automatic type conversion to
bool
```

```
        void flip() noexcept;             // bit complement
    };
    ...
    // operations for element access return reference proxy instead of bool:
    reference operator[](size_type idx);
    reference at(size_type idx);
    reference front();
    reference back();
    ...
  };
}
```

As you can see, all member functions for element access return type `reference`. Thus, you could program something like the following statements:

```
c.front().flip();      // negate first Boolean element
c[5] = c.back();       // assign last element to element with index 5
```

As usual, to avoid undefined behavior, the caller must ensure that the first, sixth, and last elements exist here.

Note that the internal proxy type `reference` is used only for nonconstant containers of type `vector<bool>`. The constant member functions for element access return values of type `const_reference`, which is a type definition for `bool`.