Vectors

- At the end of this topic the student will be able to perform the following with Vectors
  - Create
  - Resize
  - Determine when to use Vectors
  - Determine the size and capacity
  - Add elements
  - Resize
  - Iterate forwards and reverse both const and non-const
  - Shrink
  - Manipulate elements
  - Copy contents
  - Use list initializers
  - Access elements
  - Insert elements
  - Erase elements
  - Clear elements
- Iterate over built in arrays with pointers

- The Standard Library provides scores of *algorithms* you'll use frequently to manipulate a variety of containers
  - *Inserting*, *deleting*, *searching*, *sorting* and others are appropriate for some or all of the sequence and associative containers

- The *algorithms operate on container elements only indirectly through iterators*
- Many algorithms operate on sequences of elements defineds by iterators
  - Pointing to the *first element* of the sequence
  - And to *one element past the last element*

- *Sequence containers*
  - `Array`
  - `Vector`
  - `Deque`
  - `List`
  - `forward_list`
- Class templates `array`, `vector` and `deque`
  - Based on built-in arrays
- `list` and `forward_list`
  - Implement linked-list data structures

Software Engineering Observation 15.2

It's usually preferable to reuse Standard Library containers rather than developing customized templatized data structures. For novices, `vector` is typically satisfactory for most applications.

Performance Tip 15.2

Insertion at the back of a `vector` is efficient. The `vector` simply grows, if necessary, to accommodate the new item. It's expensive to insert (or delete) an element in the middle of a `vector`—the entire portion of the `vector` after the insertion (or deletion) point must be moved, because `vector` elements occupy contiguous cells in memory.

Performance Tip 15.3

Applications that require frequent insertions and deletions at both ends of a container normally use a `deque` rather than a `vector`. Although we can insert and delete elements at the front and back of both a `vector` and a `deque`, class `deque` is more efficient than `vector` for doing insertions and deletions at the front.

Performance Tip 15.4

Applications with frequent insertions and deletions in the middle and/or at the extremes of a container normally use a `list`, due to its efficient implementation of insertion and deletion anywhere in the data structure.

- Class template `vector`
  - Provides a data structure with *contiguous* memory locations
  - Enabling efficient, direct access to any element of a vector via the subscript operator `[]`, exactly as with a built-in array
- Like class template `array`, template `vector` is most commonly used
  - When subscripting is needed
  - Or will be sorted
  - Or when the number of elements may need to grow
- When a `vector`'s memory is exhausted
  - The `vector` *allocates* a larger built-in array
  - *Copies* (or *moves*) the original elements into the new built-in array
  - *Deallocates* the old built-in array

Performance Tip 15.5

Choose the `vector` container for the best random-access performance in a container that can grow.

Performance Tip 15.6

Objects of class template `vector` provide rapid indexed access with the overloaded subscript operator `[]` because they're stored in contiguous memory like a built-in array or an `array` object.

- Figure 15.10 illustrates several functions of the `vector` class template
  - Many of these functions are available in every *first-class container*
  - You must include header `<vector>` to use class template `vector`

- `size` and `capacity`
  - Each initially return 0
- `size`
  - Available in *every* container except `forward_list`
  - Returns the number of elements currently stored in the container
- Capacity (specific to vector and deque)
  - Returns the number of elements that can be stored in the vector before the vector needs to dynamically resize itself to accommodate more elements

- `push_back`
  - Available in *sequence* containers other than array and `forward_list`
  - Adds an element to the end of the `vector`
- If an element is added to a full `vector`
  - The `vector` increases its size
  - By default, most implementations have the `vector` *double* its capacity
- Sequence containers other than `array` and `vector` also provide a `push_front` function

Performance Tip 15.7

It can be wasteful to double a `vector`'s size when more space is needed. For example, a full `vector` of 1,000,000 elements resizes to accommodate 2,000,000 elements when a new element is added. This leaves 999,999 unused elements. You can use `resize` and `reserve` to control space usage better.

- The manner in which a `vector` grows to accommodate more elements—a time consuming operation
  - Is not specified by the C++ Standard
- C++ library implementers use various clever schemes to minimize the overhead of *resizing* a `vector`
  - The output of this program may vary, depending on the version of `vector` that comes with your compiler.
- Some library implementers allocate a large initial capacity
  - If a `vector` stores a small number of elements, such capacity may be a waste of space

- It can greatly improve performance
  - If a program adds many elements to a `vector`
  - And does not have to reallocate memory to accommodate those elements
  - Classic *space–time trade-off*
- Library implementers must balance the amount of memory used against the amount of time required to perform various `vector` operations

- You can output the contents of the built-in array `values` using pointers and pointer arithmetic
  - Pointers into a built-in array can be used as iterators
- Function `begin`
  - Returns an iterator pointing to the built-in array's first element
- Function `end`
  - Returns an iterator representing the position one element after the end of the built-in array
- Use the `!=` operator in the loop-continuation condition when comparing against **end**
  - When iterating using pointers to built-in array elements, it's common for the loop-continuation condition to test whether the pointer has reached the end of the built-in array

- You can infer a control variable's type (`vector<int>::const_iterator`) using the `auto` keyword
  - Prior to C++11, you would have used the overloaded begin member function to get the `const_iterator`
  - When called on a `const` container
    - Begin returns a `const_iterator`
  - The other version of begin returns an iterator that can be used for non-`const` containers

```
// display vector elements using const_iterator
for ( auto constIterator = integers2.cbegin();
      constIterator != integers2.cend();
      ++constIterator )
{
   cout << *constIterator << ' ';
}
```

could have been replaced with the following range-based for statement:

```
for ( auto const &item : integers2 )
{
   cout << item << ' ';
}
```

Common Programming Error 15.1

Attempting to dereference an iterator positioned outside its container is a runtime logic error. In particular, the iterator returned by `end` should not be dereferenced or incremented.

- C++11 now includes `vector` member function `crbegin` and `crend`
  - Which return `const_reverse_iterators` that represent the starting and ending points when iterating through a container in reverse

- As with functions `cbegin` and `cend`
  - Prior to C++11 you would have used the overloaded member functions `rbegin` and `rend` to obtain `const_reverse_iterators` or `reverse_iterators`, based on whether the container is `const`

- `shrink_to_fit`
  - As of C++11
  - For a vector or `deque`
    - Returns unneeded memory to the system
  - Requests that the container reduce its capacity to the number of elements in the container
- According to the C++ standard
  - Implementations can ignore this request so that they can perform implementation-specific optimizations

- In C++11, you can use list initializers to initialize vectors as in
  - `vector< int > integers{1, 2, 3, 4, 5, 6};`
- Or
  - `vector< int > integers = {1, 2, 3, 4, 5, 6};`
  - Not fully supported across all compilers yet

```
// Fig. 15.11: fig15_11.cpp
// Testing Standard Library vector class template
// element-manipulation functions.
```

- An `ostream_iterator< int >`
  - Outputs only values of type `int` or a compatible type

```
ostream_iterator<int> output(cout, " ");
```

- The first argument to the constructor specifies the output stream
- Second argument is a string specifying the separator for the values output
  - In this case, the string contains a space character

- `copy` (from header `<algorithm>`)

```
copy(integers.cbegin(), integers.cend(), output);
```

  - Copies each element in a range from the location specified by the iterator in its first argument and up to, but *not* including, the location specified by the iterator in its second argument
- These two arguments must satisfy *input iterator* requirements
  - They must be iterators through which values can be read from a container, such as `const_iterators`
- They must also represent a range of elements
  - Applying ++ to the first iterator must eventually cause it to reach the second iterator argument in the range

- `front` and `back` (available for most *sequence containers*)
  - Determine the `vector`'s first and last elements, respectively
- Function **front**
  - Returns a *reference* to the first element in the vector
- Function **begin**
  - Returns a *random access iterator* pointing to the first element in the vector.
- Function **back**
  - Returns a *reference* to the `vector`'s last element
- Function **end**
  - Returns a *random access iterator* pointing to the location *after* the last element

Common Programming Error 15.2

The `vector` must not be empty; otherwise, the results of `front` and `back` are undefined.

Exception type	Description
`out_of_range`	Indicates when subscript is out of range—e.g., when an invalid subscript is specified to `vector` member function `at`.
`invalid_argument`	Indicates an invalid argument was passed to a function.
`length_error`	Indicates an attempt to create too long a container, `string`, etc.
`bad_alloc`	Indicates that an attempt to allocate memory with `new` (or with an allocator) failed because not enough memory was available.

```
integers.insert(integers.begin() + 1,
 22);
```

- `insert` functions are provided by each *sequence container*
  - Except `array`, which has a fixed size
  - And `forward_list`, which has the function `insert_after` instead
- Inserts the value 22 before the element at the location specified by the iterator in the first argument
  - In this example, the iterator is pointing to the `vector`'s second element
  - So 22 is inserted as the second element and the original second element becomes the third element

- Other versions of `insert` allow inserting multiple copies of the same value starting at a particular position
- Or inserting a range of values from another container, starting at a particular position
- As of C++11, this version of member function insert returns an iterator pointing to the item that was inserted

- `erase` functions are available in all *first-class containers*
  - Except `array`, which has a fixed size
    - And `forward_list`, which has the function `erase_after` instead

```
integers.erase( integers.begin() );
```

  - Erases the element at the location specified by the iterator argument
    - In this example, the first element

```
integers.erase(integers.begin(), integers.end());
```

  - Specifies that all elements in the range specified by two iterator arguments should be erased.
    - In this example, all the elements are erased.

Common Programming Error 15.3

Normally `erase` destroys the objects that are erased from a container. However, erasing an element that contains a pointer to a dynamically allocated object does not `delete` the dynamically allocated memory—this can lead to a memory leak. If the element is a `unique_ptr`, the `unique_ptr` would be destroyed and the dynamically allocated memory would be deleted. If the element is a `shared_ptr`, the reference count to the dynamically allocated object would be decremented and the memory would be deleted only if the reference count reached `0`.

```
integers.insert(integers.begin(),
values.begin(), values.end());
```

- Uses the second and third arguments to specify the starting location and ending location in a sequence of values (in this case, from the `array` `values`) that should be inserted into the `vector`
- The ending location specifies the position in the sequence *after* the last element to be inserted
- Copying occurs up to, but *not* including, this location
- As of C++11, this version of member function `insert` returns an iterator pointing to the first item that was inserted
  - If nothing was inserted, the function returns its first argument

```
integers.clear();
```

- Found in all *first-class containers* except `array`
- Empties the `vector`
  - Does not necessarily return any of the `vector`'s memory to the system

- This topic showed how to and performed the following with Vectors
  - Create
  - Resize
  - Determine when to use Vectors
  - Determine the size and capacity
  - Add elements
  - Resize
  - Iterate forwards and reverse both const and non-const
  - Shrink
  - Manipulate elements
  - Copy contents
  - Use list initializers
  - Access elements
  - Insert elements
  - Erase elements
  - Clear elements
- Iterate over built in arrays with pointers