C++ Containers

Matt Warner

1 Overview of C++ containers

A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows great flexibility in the types supported as elements.

The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properites to pointers)

2 Sequence Containers

Sequence containers implement base structures that can be accessed sequentially. The STL containers that are of this type are:

- **array**: Static contiguous array
- **vector**: Dynamic contiguous array
- **deque**: double-ended queue
- **forward_list**: Singly-linked list
- **list**: Doubly-linked list

2.1 std::array

The introduction of array classes from C++11 has offered a better alternative for C-style arrays. The advantages of array class over C-style array are:

1. Array classes knows its own size, wheras C-style arrays lack this property. So when passing to functions, we don't need to pass size of Array as a seperate parameter.
2. With C-style array ther is more risk of array being decayed into a pointer. Array classes don't decay into pointers.
3. Array classes are generally more eficient, light-weight and reliable than C-style arrays.

The main drawback for using std::array is that it is a statically-sized cotainer, meaning the size of the array is defined once and remains constant throughout the lifetime of the array.

Operations on array

- **at()**: This function is used to access the elements of array
- **get():** This function is also used to access the elements of array. This function is not the member of array class but overloaded function from class tuple.
- **operator[]**: This is similar to C-style arrays. this method is also used to access array elements.
- **front()** This returns a reference to the first element in the array
- **back()** This returns a reference to the last element in the array
- **size()** returns the number of elements in array. This is property that C-style arrays lack.
- **max_size()** Returns the maxiumum number fo elements array can hold i.e, the size with which array is declared.
- **swap()** swaps element in array with another array
- **empty()** Determines if array is empty

Example:

```cpp
#include <array> // for at()
#include <tuple> // for get()

std::array<int,4> arr = {1,2,3,4};

// Using at()
for (uint8_t i = 0; i < 4; ++i) { std::cout << arr.at(i) << " ";}

// Stuff with get()
int first_element = std::get<0>(arr);
std::get<0>(arr) = 0;
std::cout << std::get<0>(arr);

// front() and back()
ar.front() = 10;
ar.back() = 10;

// Using swap()
std::array<int, 6> arr2 = {5,6,7,8};
ar.swap(arr2);

// Using empty()
if (ar.empty()) { cout << "Empty array!"}
```

Iterator category

The standard array implementation uses a **Random Access Iterator**, meaning that it can directly access any element in the container using arithmetic. std::array just uses a raw pointer, which is common with random access iterators. This is a very simple container, so not much work need to go into its iterator.

Example implementation of std::array

```cpp
#ifndef ARRAY_H
#define ARRAY_H

template <typename T, std::size_t N>
struct array {

typedef T value_type
typedef std::size_t size_type

typedef T* iterator
typedef const T* const_iterator
typedef T* std::reverse_iterator<iterator> reverse_iterator
typedef const std::reverse_iterator<const_iterator> const_reverse_iterator

value_type base[N ? N : 1 ];

void fill(value_type element) {
  std::fill_n(begin(),end(), element);
}
void swap(struct array &rhs) {
  std::swap_ranges(begin(),end(), rhs.begin());
```

```
    }

    // Element access.
    value_type& operator[](size_type &index) { return base[index]; }
    const value_type& operator[](size_type &index) const { return base[index]; }
    value_type& at(size_type index) {
      if (index > N) {
        throw(std::out_of_range("array::at"));
      }
      return base[index];
    }
    // Capacity.
    size_type size() const { return N; }
    size_type max_size() const { return N; }
    bool empty() const { return size() == 0; }

    // Iterators.
    iterator begin() { return std::addressof(base[0]); }
    const_iterator begin() const { return std::addressof(base[0]); }
    iterator end() { return std::addressof(base[N]); }
    const_iterator end() const { return std::addressof(base[N]); }
    reverse_iterator rbegin() { return std::reverse_iterator<iterator>(end()); }
    const_reverse_iterator rbegin() const { return
    ↪  std::reverse_iterator<const_iterator>(end()); }
    reverse_iterator rend() { return std::reverse_iterator<iterator>(begin());}
    const reverse_iterator rend() const { return std::reverse_iterator<iterator>(begin());}

    };
  template<typename T, typename N>
    inline bool operator==(const array<T,N> &lhs, const array<T,N> &rhs) {
      return std::equal(lhs.begin(),lhs.end(),rhs.begin());
    }
  template<typename Tp, typename std::size_t N>
  inline bool operator!=(const array<Tp,N> &lhs, const array<Tp,N> &rhs) {
      return !(lhs == rhs);
  }
  template<typename Tp, typename std::size_t N>
  inline bool operator<(const array<Tp, N> &lhs, const array<Tp, N> &rhs) {
      return std::lexicographical_compare(lhs.begin(),lhs.end(), rhs.begin(),rhs.end());
  }
  template<typename Tp, typename std::size_t N>
  inline bool operator>(const array<Tp, N> &lhs, const array<Tp, N> &rhs) {
    return (rhs < lhs);
  }
    #endif
```

Note:-

This implementation has no constructor/destructor, no assignment operators, no private or protected base members, and has no base class or virtual functions. This allows for **aggregate initialization**

In C++ and later, aggregate initialization is a way to initalize arrays and aggregate types using brace-enclosed lists.

2.2 vector

Vectors are sequence containers representing arrays that can change size

Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.

Internally, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual capacity greater than the storage strictly needed to contain its elements (i.e., its size). Libraries can implement different strategies fro growth to balance between memory usage and reallocations, but in any case, reallocations should only happen at logarithmically growing intervals of size so that the insertion of individual elements at the end of the vector can provide with *amortized constant time* complexity

Therefore, compared to arrays, vectors consume more memory in exchange for the ability to manage storage and grow dynamically in an efficient way.

2.2.1 Container Properties

Sequence

Elements in sequence containers are ordered in a strict linear sequence. Individual elements are accessed by their position in this sequence.

Dynamic array

Allows direct access to any element in the sequence, even through pointer arithmetics, and provides relatively fast addition/removal of elements at the end of the sequence.

Allocator-aware

The container uses an allocator object to dynamically handle its storage needs.

2.2.2 Template parameters

```
typename T
```

Type of the elements

Only if T is guaranteed to not throw while moving, implementations can optimize to move elements instead of copying them during reallocations

Aliased as member type **vector::value_type**

```
typename Alloc
```

Type of the allocator object used to define the storage allocation model. By default, the allocator class template is used, which defines the simplest memory allocation model and is value-independent.

Aliased as member type **vector::allocator_type**

2.2.3 Member Types

Member Type	**Definition**	**Notes**
value_type	The first template parameter (T)	
allocator_type	The second template parameter (Alloc)	Defaults to: `allocator<value_type>`
reference	`value_type&`	
const_reference	`const value_type&`	
pointer	`allocator_traits<allocator_type>::pointer`	for the default allocator: `value_type*`
const_pointer	`allocator_traits<allocator_type>::const_pointer`	for the default allocator: `const value_type*`
iterator	a random access iterator to value_type	convertible to `const_iterator`
const_iterator	a random access iterator to `const value_type`	
reverse_iterator	`reverse_iterator<const_iterator>`	
const_reverse_iterator	`reverse_iterator<const_iterator>`	
difference_type	a signed integral type, identical to: `iterator_traits<iterator>::difference_type`	usually the same as `ptrdiff_t`
size_type	an unsigned integral type that can represent any non-negative value of `difference_type`	usually the same as `size_t`

Table 1: Description of typedefs

2.2.4 Member Functions

Iterators:

`begin()`: Return iterator to beginning

`end()`: Return iterator to end

`rbegin()`: Returns a reverse iterator to reverse beginning

`rend()`: Returns reverse iterator to reverse end

`cbegin()`: Returns const_iterator to beginning

`cend()`: Returns const_iterator to end

`crbegin()`: Returns const_reverse_iterator to reverse beginning

`crend()`: Returns const_reverse_iterator to reverse end

Capacity:

`size()`: Returns size

`max_size()`: Returns maximum size

`resize()`: Change size

`capacity()`: Return size of allocated storage capacity

`empty()`: Test whether vector is empty

`reserve()`: Request a change in capacity

`shrink_to_fit()`: Shrink to fit

Element access

`operator[]`: Access element

`at()`: Access element

`front()`: Access first element

`back()`: Access last element

`base()`: Access base

Modifers

`assign()`: Assign vector content

`push_back()` Add element to end

`pop_back()` Delete last element

`insert()` Insert elements

`erase()` Erase elements

`swap()` Swap content

`clear()` Clear content

`emplace()` Construct and insert element

`emplace_back()` Construct and insert element at the end

Allocator:

`get_allocator` Get allocator

Iterator category

Like `std::array`, the standard vector implementation uses a random access iterator. However, instead of using a raw pointer like we did with `std::array`, we are going to develop our own iterator type.

2.2.5 Supported actions for vector iterator

```
*i                  // returns the current value
*i  = v             // Assign value `v` to the value dererenced.
++i                 // increments the iterator (returns a reference)
--i                 // See above
i++                 // increments the iterator (but returns a reference to the original)
i--                 // See above
*i++                // returns the current value and increments the iterator
*i--                // See above

i += n              // Moves the iterator forward
i -= n              // Same as last one
```

```
i + n               // Creates a new iterator moved forward from i
n + i               // Same as last one
i - n               // Same as last one

i1 - i2             // returns the distance between the iterators.

i[n]                // returns the item n steps forward from this iterator.
i->m                // Access the member `m` referenced by i

i1 <  i2            // Compare iterators
i1 >  i2
i1 <= i2
i1 >= i2
i1 == i2
i1 != i2
```

2.2.6 Writing a vector iterator

```
#ifndef VECTOR_ITERATOR_H
#define VECTOR_ITERATOR_H
template <typename vector>
class vector_iterator {
public:
        typedef typename vector::value_type value_type;
        typedef value_type* pointer;
        typedef value_type& reference;
        typedef std::random_access_iterator_tag iterator_category;
        typedef std::ptrdiff_t difference_type;

public:
    vector_iterator(pointer ptr = nullptr) : base(ptr) {}

    // Prefix--
    vector_iterator& operator++() {
        ++base;
        return *this;
    }
    vector_iterator operator++(int) {
        vector_iterator iterator = *this;
        ++(*this);
        return iterator;
    }

    // Postfix--
    vector_iterator& operator--(){ --base; return *this; }

    vector_iterator operator--(int) const {
        vector_iterator iterator = *this;
        --(*this);
        return iterator;
    }

    // Index operator.
    reference operator[](int pos) {
        return *(base + pos);
    }
    // Arrow operator.
    pointer operator->() const {
        return base;
    }

    // Dereference operator.
    reference operator*() const {
        return *base;
    }

    // Arithmetic operators.
    vector_iterator& operator+=(difference_type n) {
        base += n;
        return *this;
    }
    vector_iterator& operator-=(difference_type n) {
        base-= n;
        return *this;
    }
    vector_iterator operator+(difference_type n) const {
        return vector_iterator(base+n);
    }
    const vector_iterator operator-(difference_type n) const {
        return vector_iterator(base - n);
    }
    difference_type operator-(const vector_iterator& rhs) {
        return base - rhs.base;
    }

    // rhs objects
    friend vector_iterator operator+(difference_type n, const vector_iterator& rhs ) {
        return vector_iterator(rhs.base + n);
    }
    friend vector_iterator operator-(difference_type n, const vector_iterator& rhs) {
        return vector_iterator(rhs.base - n);
    }

private:
        pointer base;
};

template <typename vector>
inline bool operator==(const vector_iterator<vector>& lhs, const
↪   vector_iterator<vector>& rhs) {
    return (lhs.base == rhs.base);
}
template<typename vector>
inline bool operator!=(const vector_iterator<vector>& lhs, const
↪   vector_iterator<vector>& rhs) {
    return !(lhs.base == rhs.base);
}
template<typename vector>
inline bool operator>(const vector_iterator<vector>& lhs, const vector_iterator<vector>&
↪   rhs) {
    return (lhs.base > rhs.base);
```

```
}
template<typename vector>
inline bool operator<(const vector_iterator<vector>& lhs, const vector_iterator<vector>&
↪  rhs) {
    return (lhs.base < rhs.base);
}
template<typename vector>
inline bool operator>=(const vector_iterator<vector>& lhs, const
↪  vector_iterator<vector>& rhs) {
    return (lhs.base >= rhs.base);
}
template<typename vector>
inline bool operator<=(const vector_iterator<vector>& lhs, const
↪  vector_iterator<vector>& rhs) {
    return (lhs.base <= rhs.base);
}
```

There are a few things in the iterator class that are worth mentioning. For starters, `operator++` (prefix) `operator--` (prefix), `operator+=`, `operator-=` should all return a reference to the current object to allow for **chaining operatations**. Furthermore, the prefix version of `operator++`, and `operator--` returns the value of the object after it was changed. Thus, with prefix you can just return `*this` as a reference. The postfix version is supposed to return the value *before* the value is changed, so you're forced to create a seperate object to represent that value and return it. So with postfix, you must return by value if you want to preserve the expected meaning.

Additionally, `operator+=` and `operator-=` having parameters that are of the type `vector_iterator` are not included in our overloaded operators, since they are invalid operations. Pointers represent locations in memory, and adding them or subtracting them together doesn't make sense.

Lastly, in our class definition we overloaded `operator+` and `operator-` such that the right-hand side of the operator is an integer. However, we also need to include `operator+` and `operator-` such that they accept an integer as the left-hand side operand. This is done through friend functions since overloading operators as member functions of a class require an object of that class to reside on the left-hand side of the operator. Supported actions for a random access iterator

2.2.7 Writing our vector class

Now that we have written our vector iterator, the next step is to write the actual vector class.

```
template <typename Tp>
class vector {
public:
    typedef Tp value_type;
    typedef size_t size_type;
    typedef value_type* pointer;
    typedef const value_type* const_pointer;
    typedef value_type& reference;
    typedef const value_type& const_reference;
    typedef vector_iterator<vector<value_type>> iterator;
    typedef const vector_iterator<iterator> const_iterator;
    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
    typedef ptrdiff_t difference_type;
```

At the very top of our class definition, what we have here are various typedefs that provide code readabilty and also serve various purposes. For example, if you have ever used `std::vector`, you probally have done something like this:

```
std::vector<int>::iterator = v.begin();
```

We are allowed to do this through the typedefs provided by the `std::vector` container.

The next thing to do when implementing our vector container is to create the constructors. The vector container has many constructors for various situations. The following block of code covers the vast majority of constructors:

```cpp
public:
    // Constructors.
    vector(size_type sz = 0) : m_size(0), m_capacity(sz), data(nullptr) {}

    vector(size_type sz, value_type value) : m_size(sz) m_capacity(sz), data = new
    ↪  value_type[sz] {
        std::fill(data,data+m_size, value);
    }
    vector(std::initializer_list<Tp> l) : m_size(l.size()), m_capacity(l.size()) {
        if (m_capacity != 0) {
            data = new value_type[m_capacity];
            std::copy(l.begin(), l.end(), data);
        } else { data = nullptr; }
    }
    vector(const vector& rhs) : m_size(rhs.m_size), m_capacity(rhs.m_capacity) {
        if (m_capacity != 0) {
            data = new value_type[m_capacity];
            std::copy(rhs.data, rhs.data+rhs.m_size, data);
        } else { data = nullptr; }
    }
    vector(vector&& rhs) noexcept : m_size(rhs.m_size), m_capacity(rhs.m_capacity),
    ↪  data(rhs.data) {
            rhs.m_size = 0;
            rhs.m_capacity = 0;
            rhs.data = nullptr;
    }
    vector& operator=(const vector& rhs) {
        if (this != &rhs) {
            delete[] data;
            m_size = rhs.m_size;
            m_capacity = rhs.m_capacity;

            data = new value_type[m_capacity];
            std::copy(rhs.data, rhs.data+rhs.m_size, data);
        }
        return *this;
    }
    vector& operator=(vector&& rhs) noexcept {
        if (this != &rhs) {
            delete[] data;
            m_size = rhs.m_size;
            m_capacity = rhs.m_capacity;
            data = rhs.data;

            rhs.m_size = 0;
            rhs.m_capacity = 0;
            rhs.data = nullptr;
        }
        return *this;
    }
    ~vector() {
        delete[] data;
    }
```

A good vector implementation supports *move constructors* and *move assignment operators* as a means to move one vector container to another without having to perform any copying. Additionally, its important when designing your constructors to provide one that takes a `std::initalizer_list`. This is so you can initalize a vector as such:

```
vector<int> v = {1,2,3,4};
vector<int> v({1,2,3,4}); // also valid;
```

`initalizer_list`'s are a fairly interesting feature of C++ and behave in ways that you might not expect them to. For instance, although we dont have a assignment operator overload that takes an std::initalizer_list, the following statement is entirely valid:

```
vector<int> v = {1,2,3,4};
v = {5,6,7,8};
```

This is what the compiler is doing with our code, and the reason why this works:

```
vector<int> v = vector<int>{std::initializer_list<int>{1,2,3,4}};
v.operator=(vector<int>{std::initializer_list<int>{5,6,7,8}});
```

So, the compiler is creating a temporary object of type `vector<int>`, resulting in a call to the constructor that takes an `initializer_list`. It then uses that object in the subsequent call to `operator=`. If you are confused as to why we can pass an r-value to `oeartor=`, which expects a l-value reference, its due to the fact that `const` l-values can bind to r-values. If `operator=` took a non-const object, then the code would not work.

Next on the list is our capacity member functions.

```
        // Capacity.
        size_type size() const { return m_size; }
        size_type max_size const { return std::numerical_limits<size_t>::max(); }
        size_type capacity const { return m_capacity; }
        bool empty() { return m_size == 0;}
        void resize(size_type sz) {
           if (sz > m_size ) {
                if (sz > m_capacity) {
                    reserve(std::max(sz, (2*m_capacity)));
                }
                size_type n_fill = sz - m_size;
                std::fill_n(n_fill, value_type());
                m_size = sz;
           }
           if (sz < m_size) {
                for (size_t i = m_size; i > sz; --i) {
                    m_data[i].~value_type();
                }
                m_size = sz;
           }
        }
    private:
        size_type m_size;
        size_type m_capacity;
        pointer data;
    };
```