16.x — Chapter 16 summary and quiz

ALEX JANUARY 17, 2024

Words of encouragement

This chapter isn't an easy one. We covered a lot of material, and unearthed some of C++'s warts. Congrats on making it through!

Arrays are one of the keys that unlock a huge amount of power within your C++ programs.

Chapter Review

A **container** is a data type that provides storage for a collection of unnamed objects (called **elements**). We typically use containers when we need to work with a set of related values.

The number of elements in a container is often called it's **length** (or sometimes **count**). In C++, the term **size** is also commonly used for the number of elements in a container. In most programming languages (including C++), containers are **homogenous**, meaning the elements of a container are required to have the same type.

The **Containers library** is a part of the C++ standard library that contains various class types that implement some common types of containers. A class type that implements a container is sometimes called a **container class**.

An **array** is a container data type that stores a sequence of values **contiguously** (meaning each element is placed in an adjacent memory location, with no gaps). Arrays allow fast, direct access to any element.

C++ contains three primary array types: (C-style) arrays, the `std::vector` container class, and the `std::array` container class.

`std::vector` is one of the container classes in the C++ standard containers library that implements an array. `std::vector` is defined in the <vector> header as a class template, with a template type parameter that defines the type of the elements. Thus, `std::vector<int>` declares a `std::vector` whose elements are of type `int`.

Containers typically have a special constructor called a **list constructor** that allows us to construct an instance of the container using an initializer list. Use list initialization with an initializer list of values to construct a container with those element values.

In C++, the most common way to access array elements is by using the name of the array along with the subscript operator (`operator[]`). To select a specific element, inside the square brackets of the subscript operator, we provide an integral value that identifies which element we

want to select. This integral value is called a **subscript** (or informally, an **index**). The first element is accessed using index 0, the second is accessed using index 1, etc... Because the indexing starts with 0 rather than 1, we say arrays in C++ are **zero-based**.

`operator[]` does not do any kind of **bounds checking**, meaning it does not check to see whether the index is within the bounds of 0 to N-1 (inclusive). Passing an invalid index to `operator[]` will result in undefined behavior.

Arrays are one of the few container types that allow **random access**, meaning every element in the container can be accessed directly and with equal speed, regardless of the number of elements in the container.

When constructing a class type object, a matching list constructor is selected over other matching constructors. When constructing a container (or any type that has a list constructor) with initializers that are not element values, use direct initialization.

```
    std::vector v1 { 5 }; // defines a 1 element vector containing value `5`.
    std::vector v2 ( 5 ); // defines a 5 element vector where elements are value-initialized.
```

`std::vector` can be made const but not constexpr.

Each of the standard library container classes defines a nested typedef member named `size_type` (sometimes written as `T::size_type`), which is an alias for the type used for the length (and indices, if supported) of the container. `size_type` is almost always an alias for `std::size_t`, but can be overridden (in rare cases) to use a different type. We can reasonably assume `size_type` is an alias for `std::size_t`.

When accessing the `size_type` member of a container class, we must scope qualify it with the fully templated name of the container class. For example, `std::vector<int>::size_type`.

We can ask a container class object for its length using the `size()` member function, which returns the length as unsigned `size_type`. In C++17, we can also use the `std::size()` non-member function.

In C++20, the `std::ssize()` non-member function, which returns the length as a large signed integral type (usually `std::ptrdiff_t`, which is the type normally used as the signed counterpart to `std::size_t`).

Accessing array elements using the `at()` member function does runtime bounds checking (and throws an exception of type `std::out_of_range` if the bounds are out of range). If the exception isn't caught, the application will be terminated.

Both `operator[]` and the `at()` member function support indexing with non-const indices. However, both expect the index to be of type `size_type`, which is an unsigned integral type. This causes sign conversion problems when the indices are non-constexpr.

An object of type `std::vector` can be passed to a function just like any other object. That means if we pass a `std::vector` by value, an expensive copy will be made. Therefore, we typically pass `std::vector` by (const) reference to avoid such copies.

We can use a function template to be able to pass a `std::vector` with any element type into a function. You can use an `assert()` to ensure the vector passed in has the correct length.

The term **copy semantics** refers to the rules that determine how copies of objects are made. When we say copy semantics are being invoked, that means we've done something that will make a copy of an object.

When ownership of data is transfered from one object to another, we say that data has been **moved**.

Move semantics refers to the rules that determine how the data from one object is moved to another object. When move semantics is invoked, any data member that can be moved is moved, and any data member that can't be moved is copied. The ability to move data instead of copying it can make move semantics more efficient than copy semantics, especially when we can replace an expensive copy with an inexpensive move.

Normally, when an object is being initialized with or assigned an object of the same type, copy semantics will be used (assuming the copy isn't elided). Move semantics will be automatically used instead when the type of the object supports move semantics, and the initializer or object being assigned from is an rvalue.

We can return move-capable types (like `std::vector` and `std::string`) by value. Such types will inexpensively move their values instead of making an expensive copy.

Accessing each element of a container in some order is called **traversal**, or **traversing** the container. Traversal is also sometimes called **iterating over** or **iterating through** the container.

Loops are often used to traverse through an array, with a loop variable being used as an index. Beware of off-by-one errors, where the loop body executes one too many or one too few times.

A **range-based for loop** (also sometimes called a **for-each loop**) allows traversal of a container without having to do explicit indexing. Favor range-based-for loops over regular for-loops when traversing containers.

Use type deduction (`auto`) with ranged-based for-loops to have the compiler deduce the type of the array element. The element declaration

should use a (const) reference whenever you would normally pass that element type by (const) reference. Consider always using `const auto&` unless you need to work with copies. This will ensure copies aren't made even if the element type is later changed.

Unscoped enumerations can be used as indices, and help provide any information about the meaning of the index.

Adding an additional "count" enumerator is useful whenever we need an enumerator that represents the array length. You can assert or static_assert that an array's length is equal to the count enumerator to ensure an array is initialized with the expected number of initializers.

Arrays where the length of the array must be defined at the point of instantiation and then cannot be changed are called **fixed-size arrays** or **fixed-length arrays**. A **dynamic array** (also called a **resizable array**) is an array whose size can be changed after instantiation. This ability to be resized is what makes `std::vector` special.

A `std::vector` can be resized after instantiation by calling the `resize()` member function with the new desired length.

In the context of a `std::vector`, **capacity** is how many elements the `std::vector` has allocated storage for, and **length** is how many elements are currently being used. We can ask a `std::vector` for its capacity via the `capacity()` member function.

When a `std::vector` changes the amount of storage it is managing, this process is called **reallocation**. Because reallocation typically requires every element in the array to be copied, reallocation is an expensive process. As a result, we want to avoid reallocation as much as reasonable.

Valid indices for the subscript operator (`operator[]`) and `at()` member function is based on the vector's length, not the capacity.

`std::vector` has a member function called `shrink_to_fit()` that requests that the vector shrink its capacity to match its length. This request is non-binding.

The order in which items are added to and removed from a stack can be described as **last-in, first-out (LIFO)**. The last plate added onto the stack will be the first plate that is removed. In programming, a **stack** is a container data type where the insertion and removal of elements occurs in a LIFO manner. This is commonly implemented via two operations named **push** and **pop**.

The `std::vector` member functions `push_back()` and `emplace_back()` will increment the length of a `std::vector`, and will cause a reallocation to occur if the capacity is not sufficient to insert the value. When pushing triggers a reallocation, `std::vector` will typically allocate some extra capacity to allow additional elements to be added without triggering another reallocation the next time an element is added.

The `resize()` member function changes the length of the vector, and the capacity (if necessary).
The `reserve()` member function changes just the capacity (if necessary)

To increase the number of elements in a `std::vector`:

- Use `resize()` when accessing a vector via indexing. This changes the length of the vector so your indices will be valid.
- Use `reserve()` when accessing a vector using stack operations. This adds capacity without changing the length of the vector.

Both `push_back()` and `emplace_back()` push an element onto the stack. If the object to be pushed already exists, `push_back()` and `emplace_back()` are equivalent. However, in cases where we are creating a temporary object for the purpose of pushing it onto the vector, `emplace_back()` can be more efficient. As of C++20, there is little reason not to favor `emplace_back()` over `push_back()`.

There is a special implementation for `std::vector<bool>` that may be more space efficient for Boolean values by similarly compacting 8 Boolean values into a byte.

`std::vector<bool>` is not a vector (it is not required to be contiguous in memory), nor does it hold `bool` values (it holds a collection of bits), nor does it meet C++'s definition of a container. Although `std::vector<bool>` behaves like a vector in most cases, it is not fully compatible with the rest of the standard library. Code that works with other element types may not work with `std::vector<bool>`. As a result, `std::vector<bool>` should generally be avoided.

Quiz time

Question #1

Write definitions for the following. Use CTAD where possible (13.12 -- Class template argument deduction (CTAD) and deduction guides (https://www.learncpp.com/cpp-tutorial/class-template-argument-deduction-ctad-and-deduction-guides/)).

a) A `std::vector` initialized with the first 6 even numbers.

Show Solution (javascript:void(0))

b) A constant `std::vector` initialized with the values `1.2`, `3.4`, `5.6`, and `7.8`.

Show Solution (javascript:void(0))

c) A constant `std::vector` of `std::string_view` initialized with the names "Alex", "Brad", "Charles", and "Dave".

Show Solution (javascript:void(0))

d) A `std::vector` with the single element value `12`.

Show Solution (javascript:void(0))

e) A `std::vector` with 12 int elements, initialized to the default values.

Show Hint (javascript:void(0))

Show Solution (javascript:void(0))

Question #2

Let's say you're writing a game where the player can hold 3 types of items: health potions, torches, and arrows.

> Step #1

Define an unscoped enum in a namespace to identify the different types of items. Define an `std::vector` to store the number of each item type the player is carrying. The player should start with 1 health potion, 5 torches, and 10 arrows. Assert to make sure the array has the correct number of initializers.

Hint: Define a count enumerator and use it inside the assert.

The program should output the following:

```
You have 16 total items
```

Show Solution (javascript:void(0))

> Step #2

Modify your program from the prior step so it now outputs:

```
You have 1 health potion
You have 5 torches
You have 10 arrows
You have 16 total items
```

Use a loop to print out the number of items and the item names for each inventory item. Handle proper pluralization of the names.

Show Solution (javascript:void(0))

Question #3

Write a function that takes a `std::vector`, returns a `std::pair` containing the indices of the elements with the min and max values in the array. The documentation for `std::pair` can be found here (https://en.cppreference.com/w/cpp/utility/pair). Call the function on the following two vectors:

```
std::vector v1 { 3, 8, 2, 5, 7, 8, 3 };
std::vector v2 { 5.5, 2.7, 3.3, 7.6, 1.2, 8.8, 6.6 };
```

The program should output the following:

```
With array ( 3, 8, 2, 5, 7, 8, 3 ):
The min element has index 2 and value 2
The max element has index 1 and value 8

With array ( 5.5, 2.7, 3.3, 7.6, 1.2, 8.8, 6.6 ):
The min element has index 4 and value 1.2
The max element has index 5 and value 8.8
```

Show Solution (javascript:void(0))

Question #4

Modify the prior program so that the user can enter as many integers as they like. Stop accepting input when the user enters `-1`.

Print the vector and find the min and max elements.

When run with the input `3 8 5 2 3 7 -1`, the program should produce the following output:

```
Enter numbers to add (use -1 to stop): 3 8 5 2 3 7 -1
With array ( 3, 8, 5, 2, 3, 7 ):
The min element has index 3 and value 2
The max element has index 1 and value 8
```

Do something reasonable when the user enters `-1` as the first input.

Show Solution (javascript:void(0))

Question #5

Let's implement the game C++man (which will be our version of the classic children's lynching game Hangman (https://en.wikipedia.org/wiki/Hangman_(game%29).

In case you've never played it before, here are the abbreviated rules:

High level:

- The computer will pick a word at random and draw an underscore for each letter in the word.
- The player wins if they guess all the letters in the word before making X wrong guesses (where X is configurable).

Each turn:

- The player will guess a single letter.
- If the player has already guessed that letter, it doesn't count, and play continues.
- If any of the underscores represent that letter, those underscores are replaced with that letter, and play continues.
- If no underscores represent that letter, the player uses up one of their wrong guesses.

Status:

- The player should know how many wrong guesses they have left.
- The player should know what letters they have guessed incorrectly.

Because this is C++man, we'll use a `+` symbol to indicate the number of wrong guesses left. If you run out of `+` symbols, you lose.

Here's sample output from the finished game:

```
Welcome to C++man (a variant of Hangman)
To win: guess the word.  To lose: run out of pluses.

The word: _________   Wrong guesses: ++++++
Enter your next letter: a
No, 'a' is not in the word!

The word: _________   Wrong guesses: +++++a
Enter your next letter: b
Yes, 'b' is in the word!

The word: b________   Wrong guesses: +++++a
Enter your next letter: c
Yes, 'c' is in the word!

The word: b__cc___   Wrong guesses: ++++ad
Enter your next letter: d
No, 'd' is not in the word!

The word: b__cc___   Wrong guesses: ++++ad
Enter your next letter: %
That wasn't a valid input.  Try again.

The word: b__cc___   Wrong guesses: ++++ad
Enter your next letter: d
You already guessed that.  Try again.

The word: b__cc___   Wrong guesses: ++++ad
Enter your next letter: e
No, 'e' is not in the word!

The word: b__cc___   Wrong guesses: +++ade
Enter your next letter: f
No, 'f' is not in the word!

The word: b__cc___   Wrong guesses: ++adef
Enter your next letter: g
No, 'g' is not in the word!

The word: b__cc___   Wrong guesses: +adefg
Enter your next letter: h
No, 'h' is not in the word!

The word: b__cc___   Wrong guesses: adefgh
You lost!  The word was: broccoli
```

> Step #1

Goals:

- We'll start by defining our list of words and writing a random word picker. You can use the Random.h from lesson 8.15 -- Global random numbers (Random.h) to assist.

Tasks:

- First define a namespace named `WordList`. The starter list of words is: "mystery", "broccoli" , "account", "almost", "spaghetti", "opinion", "beautiful", "distance", "luggage". You can add others if you like.
- Write a function to pick a random word and display the word picked. Run the program several times to make sure the word is randomized.

Here is a sample output from this step:

```
Welcome to C++man (a variant of Hangman)
To win: guess the word.  To lose: run out of pluses.

The word is: broccoli
```

Show Solution (javascript:void(0))

> Step #2

As we develop complex programs, we want to work incrementally, adding one or two things at a time and then making sure they work. What makes sense to add next?

Goals:

- Be able to draw the basic state of the game, showing the word as underscores.
- Accept a letter of input from the user, with basic error validation.

In this step, we will not yet keep track of which letters the user has entered.

Here is the sample output from this step:

```
Welcome to C++man (a variant of Hangman)
To win: guess the word.  To lose: run out of pluses.

The word: _________
Enter your next letter: %
That wasn't a valid input.  Try again.
Enter your next letter: a
You entered: a
```

Tasks:

- Create a class named `Session` that will be used to store all of the data the game needs to manage in a game session. For now, we just need to know what the random word is.
- Create a function to display the basic state of the game, where the word is displayed as underscores.
- Create a function to accept a letter of input from the user. Do basic input validation to filter out non-letters or extraneous input.

Show Solution (javascript:void(0))

> Step #3

Now that we can display some game state and get input from the user, let's integrate that user input into the game.

Goals:

- Keep track of which letters the user has guessed.
- Display correctly guessed letters.
- Implement a basic game loop.

Tasks:

- Update the Session class to track which letters have been guessed so far.
- Modify the game state function to display both underscores and correctly guessed letters.
- Update the input routine to reject letters that have already been guessed.
- Write a loop that executes 6 times before quitting (so we can test the above).

In this step, we will not tell the user whether the letter they guessed is in the word (but we will show it as part of the game state display).

The tricky part of this step is deciding how to store information on which letters the user has guessed. There are several different viable ways to do this. A hint: there are a fixed number of letters, and you're going to be doing this a lot.

Show Hint (javascript:void(0))

Show Hint (javascript:void(0))

Show Hint (javascript:void(0))

Here's sample output for this step:

```
Welcome to C++man (a variant of Hangman)
To win: guess the word.  To lose: run out of pluses.

The word: _________
Enter your next letter: a

The word: ____a___
Enter your next letter: a
You already guessed that.  Try again.
Enter your next letter: b

The word: ____a___
Enter your next letter: c

The word: ____a___
Enter your next letter: d

The word: d___a___
Enter your next letter: e

The word: d___a__e
Enter your next letter: f

The word: d___a__e
Enter your next letter: g
```

Show Solution (javascript:void(0))

> Step #4

Goal: Finish the game.

Tasks:

- Add in display of total wrong guesses left
- Add in display of incorrect letters guessed
- Add in win/loss condition

Show Solution (javascript:void(0))

Next lesson
17.1 Introduction to std::array

Back to table of contents

Previous lesson
16.12 std::vector<bool>

Avatars from https://gravatar.com/ are connected to your provided email address.

49 COMMENTS

Newest