17.x — Chapter 17 summary and quiz

Chapter Review

Fixed-size arrays (or **fixed-length arrays**) require that the length of the array be known at the point of instantiation, and that length cannot be changed afterward. C-style arrays and `std::array` are both fixed-size arrays. Dynamic arrays can be resized at runtime. `std::vector` is a dynamic array.

The length of a `std::array` must be a constant expression. Most often, the value provided for the length will be an integer literal, constexpr variable, or an unscoped enumerator.

`std::array` is an aggregate. This means it has no constructors, and instead is initialized using aggregate initialization.

Define your `std::array` as constexpr whenever possible. If your `std::array` is not constexpr, consider using a `std::vector` instead.

Use class template argument deduction (CTAD) to have the compiler deduce the type and length of a std::array from its initializers.

`std::array` is implemented as a template struct whose declaration looks like this:

```
template<typename T, std::size_t N> // N is a non-type template parameter
struct array;
```

The non-type template parameter representing the array length (`N`) has type `std::size_t`.

To get the length of a `std::array`:

- We can ask a `std::array` object for its length using the `size()` member function (which returns the length as unsigned `size_type`).
- In C++17, we can use the `std::size()` non-member function (which for `std::array` just calls the `size()` member function, thus returning the length as unsigned `size_type`).
- In C++20, we can use the `std::ssize()` non-member function, which returns the length as a large *signed* integral type (usually`std::ptrdiff_t`).

All three of these functions will return the length as a constexpr value, except when called on a `std::array` passed by reference. This defect has been addressed in C++23 by P2280.

To index a `std::array`:

- Use the subscript operator (`operator[]`). No bounds checking is done in this case, and passing in an invalid index will result in undefined behavior.
- Use the `at()` member function that does subscripting with runtime bounds checking. We recommend avoiding this function since we typically want to do bounds checking before indexing, or we want compile-time bounds checking.
- Use the `std::get()` function template, which takes the index as a non-type template argument, and does compile-time bounds checking.

You can pass `std::array` with different element types and lengths to a function using a function template with the template parameter declaration `template <typename T, std::size_t N>`. Or in C++20, use `template <typename T, auto N>`.

Returning a `std::array` by value will make a copy of the array and all elements, but this may be okay if the array is small and the elements aren't expensive to copy. Using an out parameter instead may be a better choice in some contexts.

When initializing a `std::array` with a struct, class, or array and not providing the element type with each initializer, you'll need an extra pair of braces so that the compiler will properly interpret what to initialize. This is an artifact of aggregate initialization, and other standard library container types (that use list constructors) do not require the double braces in these cases.

Aggregates in C++ support a concept called **brace elision**, which lays out some rules for when multiple braces may be omitted. Generally, you can omit braces when initializing a `std::array` with scalar (single) values, or when initializing with class types or arrays where the type is explicitly named with each element.

You can not have an array of references, but you can have an array of `std::reference_wrapper`, which behaves like a modifiable lvalue reference.

There are a few things worth noting about `std::reference_wrapper`:

- `Operator=` will reseat a `std::reference_wrapper` (change which object is being referenced).
- `std::reference_wrapper<T>` will implicitly convert to `T&`.
- The `get()` member function can be used to get a `T&`. This is useful when we want to update the value of the object being referenced.

The `std::ref()` and `std::cref()` functions were provided as shortcuts to create `std::reference_wrapper` and `const std::reference_wrapper` wrapped objects.

Use `static_assert` whenever possible to ensure a `constexpr std::array` using CTAD has the correct number of initializers.

C-style arrays were inherited from the C language, and are built-in to the core language of C++. Because they are part of the core language, C-style arrays have their own special declaration syntax. In an C-style array declaration, we use square brackets ([]) to tell the compiler that a declared object is a C-style array. Inside the square brackets, we can optionally provide the length of the array, which is an integral value of type std::size_t that tells the compiler how many elements are in the array. The length of a C-style array must be a constant expression.

C-style arrays are aggregates, which means they can be initialized using aggregate initialization. When using an initializer list to initialize all elements of a C-style array, it's preferable to omit the length and let the compiler calculate the length of the array.

C-style arrays can be indexed via `operator[]`. The index of a C-style array can be either a signed or an unsigned integer, or an unscoped enumeration. This means that C-style arrays are not subject to all of the sign conversion indexing issues that the standard library container classes have!

C-style arrays can be const or constexpr.

To get the length of a C-style array:

- In C++17, we can use the `std::size()` non-member function, which returns the length as unsigned `std::size_t`.
- In C++20, we can use the `std::ssize()` non-member function, which returns the length as a large *signed* integral type (usually`std::ptrdiff_t`).

In most cases, when a C-style array is used in an expression, the array will be implicitly converted into a pointer to the element type, initialized with the address of the first element (with index 0). Colloquially, this is called **array decay** (or just decay for short).

Pointer arithmetic is a feature the allows us to apply certain integer arithmetic operators (addition, subtraction, increment, or decrement) to a pointer to produce a new memory address. Given some pointer `ptr`, `ptr + 1` returns the address of the next *object* in memory (based on the type being pointed to).

Use subscripting when indexing from the start of the array (element 0), so the array indices line up with the element.
Use pointer arithmetic when doing relative positioning from a given element.

C-style strings are just C-style arrays whose element type is `char` or `const char`. As such, C-style strings will decay.

The **dimension** of an array is the number of indices needed to select an element.

An array containing only a single dimension is called a **single-dimensional array** or a **one-dimensional array** (sometimes abbreviated as a **1d array**). An array of arrays is called a **two-dimensional array** (sometimes abbreviated as a **2d array**) because it has two subscripts. Arrays with more than one dimension are called **multidimensional arrays**. **Flattening** an array is a process of reducing the dimensionality of an array (often down to a single dimension).

In C++23, `std::mdspan` is a view that provides a multidimensional array interface for a contiguous sequence of elements.

Quiz time

Question #1

What's wrong with each of these snippets, and how would you fix it?

a)

```
#include <array>
#include <iostream>

int main()
{
    std::array arr { 0, 1, 2, 3 };

    for (std::size_t count{ 0 }; count <= std::size(arr); ++count)
    {
        std::cout << arr[count] << ' ';
    }

    std::cout << '\n';

    return 0;
}
```

Show Solution

b)

```
#include <iostream>

void printArray(int array[])
{
    for (int element : array)
    {
        std::cout << element << ' ';
    }
}

int main()
{
    int array[] { 9, 7, 5, 3, 1 };

    printArray(array);

    std::cout << '\n';

    return 0;
}
```

Show Solution

c)

```
#include <array>
#include <iostream>

int main()
{
    std::cout << "Enter the number of test scores: ";
    std::size_t length{};
    std::cin >> length;

    std::array<int, length> scores;

    for (std::size_t i { 0 } ; i < length; ++i)
    {
        std::cout << "Enter score " << i << ": ";
        std::cin >> scores[i];
    }
    return 0;
}
```

Show Solution

Question #2

In this quiz, we're going to implement Roscoe's potion emporium, the finest potion shop in the land! This is going to be a bigger challenge.

Implement a program that outputs the following:

```
Welcome to Roscoe's potion emporium!
Enter your name: Alex
Hello, Alex, you have 85 gold.

Here is our selection for today:
0) healing costs 20
1) mana costs 30
2) speed costs 12
3) invisibility costs 50
Enter the number of the potion you'd like to buy, or 'q' to quit: a
That is an invalid input.  Try again: 3
You purchased a potion of invisibility.  You have 35 gold left.

Here is our selection for today:
0) healing costs 20
1) mana costs 30
2) speed costs 12
3) invisibility costs 50
Enter the number of the potion you'd like to buy, or 'q' to quit: 4
That is an invalid input.  Try again: 2
You purchased a potion of speed.  You have 23 gold left.

Here is our selection for today:
0) healing costs 20
1) mana costs 30
2) speed costs 12
3) invisibility costs 50
Enter the number of the potion you'd like to buy, or 'q' to quit: 2
You purchased a potion of speed.  You have 11 gold left.

Here is our selection for today:
0) healing costs 20
1) mana costs 30
2) speed costs 12
3) invisibility costs 50
Enter the number of the potion you'd like to buy, or 'q' to quit: 4
You can not afford that.

Here is our selection for today:
0) healing costs 20
1) mana costs 30
2) speed costs 12
3) invisibility costs 50
Enter the number of the potion you'd like to buy, or 'q' to quit: q

Your inventory contains:
2x potion of speed
1x potion of invisibility
You escaped with 11 gold remaining.

Thanks for shopping at Roscoe's potion emporium!
```

The player starts with a randomized amount of gold, between 80 and 120.

Sound fun? Let's do it! Because this will be hard to implement all at once, we'll develop this in steps.

> Step #1

Create a `Potion` namespace containing an enum named `Type` containing the potion types. Create two `std::array`: an `int` array to hold the potion costs, and a `std::string_view` array to hold the potion names.

Also write a function named `shop()` that enumerates through the list of `Potions` and prints their numbers, names, and cost.

The program should output the following:

```
Here is our selection for today:
0) healing costs 20
1) mana costs 30
2) speed costs 12
3) invisibility costs 50
```

Show Hint

Show Solution

> Step #2

Create a `Player` class to store the player's name, potion inventory, and gold. Add the introductory and goodbye text for Roscoe's emporium. Get the player's name and randomize their gold.

Use the "Random.h" file in lesson 8.15 -- Global random numbers (Random.h) to make randomization easy.

The program should output the following:

```
Welcome to Roscoe's potion emporium!
Enter your name: Alex
Hello, Alex, you have 84 gold.

Here is our selection for today:
0) healing costs 20
1) mana costs 30
2) speed costs 12
3) invisibility costs 50

Thanks for shopping at Roscoe's potion emporium!
```

Show Solution

> Step #3

Add the ability to purchase potions, handling invalid input (treat any extraneous input as a failure). Print the player's inventory after they leave. The program should be complete after this step.

Make sure you test for the following cases:

- User enters an invalid potion number (e.g. 'd')
- User enters a valid potion number but with extraneous input (e.g. `2d`, `25`)

We cover invalid input handling in lesson 9.5 -- std::cin and handling invalid input.

Show Hint

Show Hint

Show Solution

Question #3

Let's say we want to write a card game that uses a standard deck of cards. In order to do that, we're going to need some way to represent those cards, and decks of cards. Let's build that functionality.

We'll use it in the next quiz question to actually implement a game.

> Step #1

A deck of cards has 52 unique cards (13 card ranks of 4 suits). Create enumerations for the card ranks (ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, jack, queen, king) and suits (clubs, diamonds, hearts, spades).

Show Solution

> Step #2

Each card will be represented by a struct named `Card` that contains a rank and a suit member. Create the struct and move the enums into it.

Show Solution

> Step #3

Next, let's add some useful functions to our Card struct. First, overload `operator<<` to print the card rank and suit as a 2-letter code (e.g. the jack of spades would print as JS). You can do that by completing the following function:

```
struct Card
{
    // Your other stuff here

    friend std::ostream& operator<<(std::ostream& out, const Card &card)
    {
        out << // print your card rank and suit here
        return out;
    }
};
```

Second, add a function that returns the value of the Card. Treat an ace as value 11. Finally, add a `std::array` of Rank and of Suit (named `allRanks` and `allSuits` respectively) so they can be iterated over. Because these are part of a struct (not a namespace), make them static so they are only instantiated once (not with each object).

The following should compile:

```
int main()
{
    // Print one card
    Card card { Card::rank_5, Card::suit_heart };
    std::cout << card << '\n';

    // Print all cards
    for (auto suit : Card::allSuits)
        for (auto rank : Card::allRanks)
            std::cout << Card { rank, suit } << ' ';
    std::cout << '\n';

    return 0;
}
```

and produce the following output:

```
5H
AC 2C 3C 4C 5C 6C 7C 8C 9C TC JC QC KC AD 2D 3D 4D 5D 6D 7D 8D 9D TD JD QD KD AH 2H
3H 4H 5H 6H 7H 8H 9H TH JH QH KH AS 2S 3S 4S 5S 6S 7S 8S 9S TS JS QS KS
```

Show Solution

> Step #4

Next, let's create our deck of cards. Create a class named `Deck` that contains a `std::array` of Cards. You can assume a deck is 52 Cards.

The Deck should have three functions:

First, the default constructor should initialize the array of cards. You can use a ranged-for loop similar to the one in the main() function of the prior example to traverse through all the suits and ranks.

Second, add a `dealCard()` function that returns the next card in the Deck by value. Since `std::array` is a fixed-size array, think about how you will keep track of where the next card is. This function should assert out if it is called when the Deck has gone through all the cards.

Third, write a `shuffle()` member function that shuffles the deck. To make this easy, we will enlist the help of `std::shuffle`:

```
#include <algorithm> // for std::shuffle
#include "Random.h"  // for Random::mt

    // Put this line in your shuffle function to shuffle m_cards using the Random::mt
Mersenne Twister
    // This will rearrange all the Cards in the deck randomly
    std::shuffle(m_cards.begin(), m_cards.end(), Random::mt);
```

The `shuffle()` function should also reset however you are tracking where the next card is back to the start of the deck.

The following program should run:

```
int main()
{
    Deck deck{};
    std::cout << deck.dealCard() << ' ' << deck.dealCard() << ' ' << deck.dealCard()
<< '\n';

    deck.shuffle();
    std::cout << deck.dealCard() << ' ' << deck.dealCard() << ' ' << deck.dealCard()
<< '\n';

    return 0;
}
```

and produce the following output (the last 3 cards should be randomized):

```
AC 2C 3C
2H 7H 9C
```

Show Solution

Question #4

Alright, now let's use our Card and Deck to implement a simplified version of Blackjack! If you're not already familiar with Blackjack, the Wikipedia article for Blackjack has a summary.

Here are the rules for our version of Blackjack:

- The dealer gets one card to start (in real life, the dealer gets two, but one is face down so it doesn't matter at this point).
- The player gets two cards to start.
- The player goes first.
- A player can repeatedly "hit" or "stand".
- If the player "stands", their turn is over, and their score is calculated based on the cards they have been dealt.
- If the player "hits", they get another card and the value of that card is added to their total score.
- An ace normally counts as a 1 or an 11 (whichever is better for the total score). For simplicity, we'll count it as an 11 here.
- If the player goes over a score of 21, they bust and lose immediately.
- When the player is done, it is the dealer's turn.
- The dealer repeatedly draws until they reach a score of 17 or more, at which point they must stop drawing.
- If the dealer goes over a score of 21, they bust and the player wins immediately.
- Otherwise, if the player has a higher score than the dealer, the player wins. Otherwise, the player loses (we'll consider ties as dealer wins for simplicity).

In our simplified version of Blackjack, we're not going to keep track of which specific cards the player and the dealer have been dealt. We'll only track the sum of the values of the cards they have been dealt for the player and dealer. This keeps things simpler.

Start with the code you wrote in the prior quiz (or use our reference solution).

> Step #1

Create a struct named `Player` that will represent a participant in our game (either the dealer or the player). Since in this game we only care about a player's score, this struct only needs one member.

Write a function that will (eventually) play a round of Blackjack. For now, this function should draw one randomized card for the dealer and two randomized cards for the player. It should return a bool value indicating who has the greater score.

The code should output the following:

```
The dealer is showing: 10
You have score: 13
You win!
```

```
The dealer is showing: 10
You have score: 8
You lose!
```

Show Solution

> Step #2

Add a `Settings` namespace that contains two constants: the value above which the player busts, and the value where the dealer must stop drawing cards.

Add the logic that handles the dealer's turn. The dealer will draw cards until they hit 17, then they must stop. If they bust, the player wins.

Here is some sample output:

```
The dealer is showing: 8
You have score: 9
The dealer flips a 4D.  They now have: 12
The dealer flips a JS.  They now have: 22
The dealer went bust!
You win!

The dealer is showing: 6
You have score: 13
The dealer flips a 3D.  They now have: 9
The dealer flips a 3H.  They now have: 12
The dealer flips a 9S.  They now have: 21
You lose!

The dealer is showing: 7
You have score: 21
The dealer flips a JC.  They now have: 17
You win!
```

Show Solution

> Step #3

Finally, add logic for the player's turn. This will complete the game.

Here is some sample output:

```
The dealer is showing: 2
You have score: 14
(h) to hit, or (s) to stand: h
You were dealt KH.  You now have: 24
You went bust!
You lose!
```

```
The dealer is showing: 10
You have score: 9
(h) to hit, or (s) to stand: h
You were dealt TH.  You now have: 19
(h) to hit, or (s) to stand: s
The dealer flips a 3D.  They now have: 13
The dealer flips a 7H.  They now have: 20
You lose!

The dealer is showing: 7
You have score: 12
(h) to hit, or (s) to stand: h
You were dealt 7S.  You now have: 19
(h) to hit, or (s) to stand: h
You were dealt 2D.  You now have: 21
(h) to hit, or (s) to stand: s
The dealer flips a 6H.  They now have: 13
The dealer flips a QC.  They now have: 23
The dealer went bust!
You win!
```

Show Solution

Question #5

a) Describe how you could modify the above program to handle the case where aces can be equal to 1 or 11.

It's important to note that we're only keeping track of the sum of the cards, not which specific cards the user has.

Show Solution

b) In actual blackjack, if the player and dealer have the same score (and the player has not gone bust), the result is a tie and neither wins. Describe how you'd modify the above program to account for this.

Show Solution

c) Extra credit: implement the above two ideas into your blackjack game. Note that you will need to show the dealer's initial card and the player's initial two cards so they know whether they have an ace or not.

Here's a sample output:

```
The dealer is showing JH (10)
You are showing AH 7D (18)
(h) to hit, or (s) to stand: h
You were dealt JD.  You now have: 18
(h) to hit, or (s) to stand: s
The dealer flips a 6C.  They now have: 16
The dealer flips a AD.  They now have: 17
You win!
```

Show Solution