10.x — Chapter 10 comprehensive quiz

ALEX SEPTEMBER 1, 2021

Words of encouragement

Congratulations on reaching the end of the longest chapter in the tutorials! Unless you have previous programming experience, this chapter was probably the most challenging one so far. If you made it this far, you're doing great!

The good news is that the next chapter is easy in comparison. And in the chapter beyond that, we reach the heart of the tutorials: Object-oriented programming!

Chapter summary

Arrays allow us to store and access many variables of the same type through a single identifier. Array elements can be accessed using the subscript operator (`[]`). Be careful not to index an array out of the array's range. Arrays can be initialized using an initializer list or uniform initialization.

Fixed arrays must have a length that is set at compile time. Fixed arrays will usually decay into a pointer when evaluated or passed to a function.

Loops can be used to iterate through an array. Beware of off-by-one errors, so you don't iterate off the end of your array. Range-based for-loops are useful when the array hasn't decayed into a pointer.

Arrays can be made multidimensional by using multiple indices.

Arrays can be used to do C-style strings. You should generally avoid these and use `std::string_view` and `std::string` instead.

Pointers are variables that store the memory address of (point at) another variable. The address-of operator (`&`) can be used to get the address of a variable. The indirection operator (`*`) can be used to get the value that a pointer points at.

A null pointer is a pointer that is not pointing at anything. Pointers can be made null by initializing or assigning the value `nullptr` to them. Avoid the `NULL` macro. Indirection through a null pointer can cause bad things to happen. Deleting a null pointer is okay (it doesn't do anything).

A pointer to an array doesn't know how large the array it is pointing to is. This means `sizeof()` and range-based for-loops won't work.

The `new` and `delete` operators can be used to dynamically allocate memory for a pointer variable or array. Although it's unlikely to happen, operator `new` can fail if the operating system runs out of memory. If you're writing software for a memory-limited system, make sure to check if `new` was successful.

Make sure to use the array delete (`delete[]`) when deleting an array. Pointers pointing to deallocated memory are called dangling pointers. Using the wrong `delete`, or indirection through a dangling pointer causes undefined behavior.

Failing to delete dynamically allocated memory can result in memory leaks when the last pointer to that memory goes out of scope.

Normal variables are allocated from limited memory called the stack. Dynamically allocated variables are allocated from a general pool of memory called the heap.

A pointer to a `const` value treats the value it is pointing to as `const`.

```
int value{ 5 };
const int *ptr{ &value }; // this is okay, ptr is pointing to a "const
int"
```

A `const` pointer is a pointer whose value can not be changed after initialization.

```
int value{ 5 };
int *const ptr{ &value }; // ptr is const, but *ptr is non-
const
```

A reference is an alias to another variable. References are declared using an ampersand (`&`), but this does not mean address-of in this context. References are implicitly `const` -- they must be initialized with a value, and a new value can not be assigned to them. References can be used to prevent copies from being made when passing data to or from a function.

The member selection operator (`->`) can be used to select a member from a pointer to a struct. It combines both an indirection and normal member access (`.`).

Void pointers are pointers that can point to any type of data. Indirection through them is not possible directly. You can use `static_cast` to convert them back to their original pointer type. It's up to you to remember what type they originally were.

Pointers to pointers allow us to create a pointer that points to another pointer.

`std::array` provides all of the functionality of C++ built-in arrays (and more) in a form that won't decay into a pointer. These should generally be preferred over built-in fixed arrays.

`std::vector` provides dynamic array functionality, handles its own memory management and remembers its size. These should generally be favored over built-in dynamic arrays.

Thanks to iterators, we don't have to know how a container is implemented to loop through its elements.

The algorithms library helps us to save a lot of time by providing many off-the-shelf functions. In combination with iterators (and later lambdas), the algorithms library is an important part of C++.

Quiz time

To make the quizzes a little easier, we have to introduce a couple of new algorithms.

`std::reduce` applies a function, by default the `+` operator, to all elements in a list, resulting in a single value. When we use the `+` operator, the result is the sum of all elements in the list. Note that there's also `std::accumulate`. `std::accumulate` cannot be parallelized, because it applies the function left-to-right. `std::reduce` segments the list, which means that the function is applied in an unknown order, allowing the operation to be parallelized. If we want to sum up a list, we don't care about the order and we use `std::reduce`.

Author's note

`std::reduce` is currently not fully implemented in all major standard libraries. If it doesn't work for you, fall back to `std::accumulate`.

`std::shuffle` takes a list and randomly re-orders its elements. We covered `std::mt19937` in lesson 9.5 -- Random number generation.

```cpp
#include <algorithm> // std::shuffle
#include <array>
#include <ctime>
#include <iostream>
#include <numeric> // std::reduce
#include <random>

int main()
{
  std::array arr{ 1, 2, 3, 4 };

  std::cout << std::reduce(arr.begin(), arr.end()) << '\n'; // 10

  // If you can't use std::reduce, use std::accumulate. The 0 is the initial value
  // of the result: (((0 + 1) + 2) + 3) + 4
  std::cout << std::accumulate(arr.begin(), arr.end(), 0) << '\n'; // 10

  std::mt19937 mt{ static_cast<std::mt19937::result_type>(std::time(nullptr)) };
  std::shuffle(arr.begin(), arr.end(), mt);

  for (int i : arr)
  {
    std::cout << i << ' ';
  }

  std::cout << '\n';

  return 0;
}
```

Possible output

```
10
10
2 1 4 3
```

Question #1

Pretend you're writing a game where the player can hold 3 types of items: health potions, torches, and arrows. Create an `enum` to identify the different types of items, and an `std::array` to store the number of each item the player is carrying (The enumerators are used as indexes of the array). The player should start with 2 health potions, 5 torches, and 10 arrows. Write a function called `countTotalItems()` that returns how many items the player has in total. Have your `main()` function print the output of `countTotalItems()` as well as the number of torches.

Show Solution

Question #2

Write the following program: Create a `struct` that holds a student's first name and grade (on a scale of 0-100). Ask the user how many students they want to enter. Create a `std::vector` to hold all of the students. Then prompt the user for each name and grade. Once the user has entered all the names and grade pairs, sort the list by grade (highest first). Then print all the names and grades in sorted order.

For the following input:

```
Joe
82
Terry
73
Ralph
4
Alex
94
Mark
88
```

The output should look like this:

```
Alex got a grade of 94
Mark got a grade of 88
Joe got a grade of 82
Terry got a grade of 73
Ralph got a grade of 4
```

You can assume that names don't contain spaces and that that input extraction doesn't fail.

Show Solution

Question #3

Write your own function to swap the value of two integer variables. Write a `main()` function to test it.

Show Hint

Show Solution

Question #4

Write a function to print a C-style string character by character. Use a pointer to step through each character of the string and print that character. Stop when you hit the null terminator. Write a `main` function that tests the function with the string literal "Hello, world!".

Show Hint

Show Solution

Question #5

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

a)

```cpp
int main()
{
  int array[]{ 0, 1, 2, 3 };

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

  std::cout << '\n';

  return 0;
}
```

Show Solution

b)

```cpp
int main()
{
  int x{ 5 };
  int y{ 7 };

  const int* ptr{ &x };
  std::cout << *ptr << '\n';
  *ptr = 6;
  std::cout << *ptr << '\n';
  ptr = &y;
  std::cout << *ptr << '\n';

  return 0;
}
```

Show Solution

c)

```cpp
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

d)

```cpp
int* allocateArray(const int length)
{
  int temp[length]{};
  return temp;
}
```

Show Solution

e)

```
int main()
{
  double d{ 5.5 };
  int* ptr{ &d };
  std::cout << ptr << '\n';

  return 0;
}
```

Show Solution

Question #6

Let's pretend we're writing a card game.

a) A deck of cards has 52 unique cards (13 card ranks of 4 suits). Create enumerations for the card ranks (2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace) and suits (clubs, diamonds, hearts, spades). Those enumerators will not be used to index arrays.

Show Solution

b) Each card will be represented by a `struct` named `Card` that contains a rank and a suit. Create the `struct`.

Show Solution

c) Create a `printCard()` function that takes a `const Card` reference as a parameter and prints the card rank and suit as a 2-letter code (e.g. the jack of spades would print as JS).

Show Hint

Show Solution

d) A deck of cards has 52 cards. Create an array (using `std::array`) to represent the deck of cards, and initialize it with one of each card. Do this in a function named `createDeck` and call `createDeck` from `main`. `createDeck` should return the deck to `main`.

Hint: Use static_cast if you need to convert an integer into an enumerated type.

Show Solution

e) Write a function named `printDeck()` that takes the deck as a `const` reference parameter and prints the cards in the deck. Use a range-based for-loop. When you can `printDeck` with the deck you generated in the previous task, the output should be

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

If you used different characters, that's fine too.

Show Solution

f) Write a function named `shuffleDeck` to shuffle the deck of cards using `std::shuffle`. Update your main function to shuffle the deck and print out the shuffled deck.

Reminder: Only seed your random number generator once.

Show Solution

g) Write a function named `getCardValue()` that returns the value of a `Card` (e.g. a 2 is worth 2, a ten, jack, queen, or king is worth 10. Assume an Ace is worth 11).

Show Solution

Question #7

a) Alright, challenge time! Let's write 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.
- The dealer goes after the player.
- The dealer repeatedly draws until they reach a score of 17 or more, at which point they stand.
- 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 quiz #6. Create a function named `playBlackjack()`. This function should:

- Accept a shuffled deck of cards as a parameter.
- Implement Blackjack as defined above.
- Returns `true` if the player won, and `false` if they lost.

Also write a `main()` function to play a single game of Blackjack.

Show Solution

Once you've solved the quiz, have a look at some of the most common mistakes:

Show Hint

b) Extra credit: Critical thinking time: 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

c) 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

Next lesson
11.1 Function parameters and arguments

Back to table of contents

Previous lesson
10.25 Introduction to standard library algorithms

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

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

Notify me about replies: POST COMMENT

L DP N N FOUT

Newest

X

X