12.x — Chapter 12 comprehensive quiz

ALEX SEPTEMBER 19, 2021

In this chapter, we explored the meat of C++ -- object-oriented programming! This is the most important chapter in the tutorial series.

Quick Summary

Classes allow you to create your own data types that bundle both data and functions that work on that data. Data and functions inside the class are called members. Members of the class are selected by using the . operator (or -> if you're accessing the member through a pointer).

Access specifiers allow you to specify who can access the members of a class. Public members can be accessed directly by anybody. Private members can only be accessed by other members of the class. We'll cover protected members later, when we get to inheritance. By default, all members of a class are private and all members of a struct are public.

Encapsulation is the process of making all of your member data private, so it can not be accessed directly. This helps protect your class from misuse.

Constructors are a special type of member function that allow you to initialize objects of your class. A constructor that takes no parameters (or has all default parameters) is called a default constructor. The default constructor is used if no initialization values are provided by the user. You should always provide at least one constructor for your classes.

Member initializer lists allows you to initialize your member variables from within a constructor (rather than assigning the member variables values).

Non-static member initialization allows you to directly specify default values for member variables when they are declared.

Constructors are allowed to call other constructors (called delegating constructors, or constructor chaining).

Destructors are another type of special member function that allow your class to clean up after itself. Any kind of deallocation or shutdown routines should be executed from here.

All member functions have a hidden *this pointer that points at the class object being modified. Most of the time you will not need to access this pointer directly. But you can if you need to.

It is good programming style to put your class definitions in a header file of the same name as the class, and define your class functions in a .cpp file of the same name as the class. This also helps avoid circular dependencies.

Member functions can (and should) be made const if they do not modify the state of the class. Const class objects can only call const member functions.

Static member variables are shared among all objects of the class. Although they can be accessed from a class object, they can also be accessed directly via the scope resolution operator.

Similarly, static member functions are member functions that have no *this pointer. They can only access static member variables.

Friend functions are functions that are treated like member functions of the class (and thus can access a class's private data directly). Friend classes are classes where all members of the class are considered friend functions.

It's possible to create anonymous class objects for the purpose of evaluation in an expression, or passing or returning a value.

You can also nest types within a class. This is often used with enums related to the class, but can be done with other types (including other classes) if desired.

Quiz time

Question #1

a) Write a class named Point2d. Point2d should contain two member variables of type double: m_x, and m_y, both defaulted to 0.0. Provide a constructor and a print function.

The following program should run:

```
#include <iostream>

int main()
{
    Point2d first{};
    Point2d second{ 3.0, 4.0
};
    first.print();
    second.print();

    return 0;
}
```

This should print:

```
Point2d(0, 0)
Point2d(3, 4)
```

Show Solution

b) Now add a member function named distanceTo that takes another Point2d as a parameter, and calculates the distance between them. Given two points (x1, y1) and (x2, y2), the distance between them can be calculated as std::sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2)). The std::sqrt function lives in header cmath.

The following program should run:

```
#include <iostream>

int main()
{
    Point2d first{};
    Point2d second{ 3.0, 4.0 };
    first.print();
    second.print();
    std::cout << "Distance between two points: " << first.distanceTo(second) <<
'\n';

    return 0;
}
```

This should print:

```
Point2d(0, 0)
Point2d(3, 4)
Distance between two points: 5
```

Show Solution

c) Change function distanceTo from a member function to a non-member friend function that takes two Points as parameters. Also rename it “distanceFrom”.

The following program should run:

```
#include <iostream>

int main()
{
    Point2d first{};
    Point2d second{ 3.0, 4.0 };
    first.print();
    second.print();
    std::cout << "Distance between two points: " << distanceFrom(first, second) <<
'\n';

    return 0;
}
```

This should print:

```
Point2d(0, 0)
Point2d(3, 4)
Distance between two points: 5
```

Show Solution

Question #2

Write a destructor for this class:

```
#include <iostream>

class HelloWorld
{
private:
 char* m_data{};

public:
 HelloWorld()
 {
  m_data = new char[14];
  const char* init{ "Hello, World!" };
  for (int i = 0; i < 14; ++i)
   m_data[i] = init[i];
 }

 ~HelloWorld()
 {
        // replace this comment with your destructor implementation
 }

 void print() const
 {
  std::cout << m_data << '\n';
 }

};

int main()
{
 HelloWorld hello{};
 hello.print();

    return 0;
}
```

Show Solution

Question #3

Let’s create a random monster generator. This one should be fun.

a) First, let’s create an enumeration of monster types named MonsterType. Include the following monster types: Dragon, Goblin, Ogre, Orc, Skeleton, Troll, Vampire, and Zombie. Add an additional max_monster_types enum so we can count how many enumerators there are.

Show Solution

b) Now, let’s create our Monster class. Our Monster will have 4 attributes (member variables): a type (MonsterType), a name (std::string), a roar (std::string), and the number of hit points (int). Create a Monster class that has these 4 member variables.

Show Solution

c) enum MonsterType is specific to Monster, so move the enum inside the class as a public declaration. When the enum is inside the class the “Monster” in “MonsterType” is redundant, it can be removed.

Show Solution

d) Create a constructor that allows you to initialize all of the member variables.

The following program should compile:

```
int main()
{
 Monster skeleton{ Monster::Type::skeleton, "Bones", "*rattle*", 4
};

 return 0;
}
```

Show Solution

e) Now we want to be able to print our monster so we can validate it’s correct. To do that, we’re going to need to write a function that converts a Monster::Type into a string. Write that function (called getTypeString()), as well as a print() member function.

The following program should compile:

```
int main()
{
 Monster skeleton{ Monster::Type::skeleton, "Bones", "*rattle*", 4
};
 skeleton.print();

 return 0;
}
```

and print:

```
Bones the skeleton has 4 hit points and says *rattle*
```

Show Solution

f) Now we can create a random monster generator. Let’s consider how our MonsterGenerator class will work. Ideally, we’ll ask it to give us a Monster, and it will create a random one for us. We don’t need more than one MonsterGenerator. This is a good candidate for a static class (one in which all functions are static). Create a static MonsterGenerator class. Create a static function named generateMonster(). This should return a Monster. For now, make it return anonymous Monster(Monster::Type::skeleton, “Bones”, “*rattle*“, 4);

The following program should compile:

```
int main()
{
 Monster m{ MonsterGenerator::generateMonster()
};
 m.print();

 return 0;
}
```

and print:

```
Bones the skeleton has 4 hit points and says *rattle*
```

Show Solution

g) Now, MonsterGenerator needs to generate some random attributes. To do that, we’ll need to make use of this handy function:

```
// Generate a random number between min and max (inclusive)
// Assumes srand() has already been called
static int getRandomNumber(int min, int max)
{
 static constexpr double fraction{ 1.0 / (static_cast<double>(RAND_MAX) + 1.0) };  // static used for
efficiency, so we only calculate this value once
 // evenly distribute the random number across our range
 return static_cast<int>(std::rand() * fraction * (max - min + 1) + min);
}
```

However, because MonsterGenerator relies directly on this function, let’s put it inside the class, as a static function.

Show Solution

h) Now edit function generateMonster() to generate a random Monster::Type (between 0 and Monster::Type::max_monster_types-1) and a random hit points (between 1 and 100). This should be fairly straightforward. Once you've done that, define two static fixed arrays of size 6 inside the function (named s_names and s_roars) and initialize them with 6 names and 6 sounds of your choice. Pick a random name and roar from these arrays.

The following program should compile:

```
#include <ctime> // for time()
#include <cstdlib> // for rand() and srand()

int main()
{
 std::srand(static_cast<unsigned int>(std::time(nullptr))); // set initial seed value to system clock
 std::rand(); // If using Visual Studio, discard first random value

 Monster m{ MonsterGenerator::generateMonster() };
 m.print();

 return 0;
}
```

Show Solution

i) Why did we declare variables s_names and s_roars as static?

Show Solution

Question #4

Okay, time for that game face again. This one is going to be a challenge. Let's rewrite the Blackjack game we wrote in a previous lesson (10.x -- Chapter 10 comprehensive quiz) using classes! Here's the full code without classes:

```
#include <algorithm> // std::shuffle
#include <array>
#include <cassert>
#include <ctime> // std::time
#include <iostream>
#include <random> // std::mt19937

enum class CardSuit
{
    club,
    diamond,
    heart,
    spade,

    max_suits
};

enum class CardRank
{
    rank_2,
    rank_3,
    rank_4,
    rank_5,
    rank_6,
    rank_7,
    rank_8,
    rank_9,
    rank_10,
    rank_jack,
    rank_queen,
    rank_king,
    rank_ace,

    max_ranks
};

struct Card
{
    CardRank rank{};
    CardSuit suit{};
};

struct Player
{
    int score{};
```

```
};

using deck_type = std::array<Card, 52>;
using index_type = deck_type::size_type;

// Maximum score before losing.
constexpr int maximumScore{ 21 };

// Minimum score that the dealer has to have.
constexpr int minimumDealerScore{ 17 };

void printCard(const Card& card)
{
    switch (card.rank)
    {
    case CardRank::rank_2:      std::cout << '2';   break;
    case CardRank::rank_3:      std::cout << '3';   break;
    case CardRank::rank_4:      std::cout << '4';   break;
    case CardRank::rank_5:      std::cout << '5';   break;
    case CardRank::rank_6:      std::cout << '6';   break;
    case CardRank::rank_7:      std::cout << '7';   break;
    case CardRank::rank_8:      std::cout << '8';   break;
    case CardRank::rank_9:      std::cout << '9';   break;
    case CardRank::rank_10:     std::cout << 'T';   break;
    case CardRank::rank_jack:   std::cout << 'J';   break;
    case CardRank::rank_queen:  std::cout << 'Q';   break;
    case CardRank::rank_king:   std::cout << 'K';   break;
    case CardRank::rank_ace:    std::cout << 'A';   break;
    default:
        std::cout << '?';
        break;
    }

    switch (card.suit)
    {
    case CardSuit::club:       std::cout << 'C';   break;
    case CardSuit::diamond:    std::cout << 'D';   break;
    case CardSuit::heart:      std::cout << 'H';   break;
    case CardSuit::spade:      std::cout << 'S';   break;
    default:
        std::cout << '?';
        break;
    }
}

int getCardValue(const Card& card)
{
  switch (card.rank)
  {
  case CardRank::rank_2:        return 2;
  case CardRank::rank_3:        return 3;
  case CardRank::rank_4:        return 4;
  case CardRank::rank_5:        return 5;
  case CardRank::rank_6:        return 6;
  case CardRank::rank_7:        return 7;
  case CardRank::rank_8:        return 8;
  case CardRank::rank_9:        return 9;
  case CardRank::rank_10:       return 10;
  case CardRank::rank_jack:     return 10;
  case CardRank::rank_queen:    return 10;
  case CardRank::rank_king:     return 10;
  case CardRank::rank_ace:      return 11;
  default:
    assert(false && "should never happen");
    return 0;
  }
}

void printDeck(const deck_type& deck)
{
    for (const auto& card : deck)
    {
        printCard(card);
        std::cout << ' ';
    }

    std::cout << '\n';
}

deck_type createDeck()
{
  deck_type deck{};
  // We could initialize each card individually, but that would be a pain.  Let's use a loop.

  index_type index{ 0 };

  for (int suit{ 0 }; suit < static_cast<int>(CardSuit::max_suits); ++suit)
  {
    for (int rank{ 0 }; rank < static_cast<int>(CardRank::max_ranks); ++rank)
    {
      deck[index].suit = static_cast<CardSuit>(suit);
      deck[index].rank = static_cast<CardRank>(rank);
      ++index;
    }
  }

  return deck;
}

void shuffleDeck(deck_type& deck)
{
    static std::mt19937 mt{ static_cast<std::mt19937::result_type>(std::time(nullptr)) };

    std::shuffle(deck.begin(), deck.end(), mt);
}

bool playerWantsHit()
{
    while (true)
    {
        std::cout << "(h) to hit, or (s) to stand: ";

        char ch{};
        std::cin >> ch;

        switch (ch)
        {
        case 'h':
            return true;
        case 's':
            return false;
        }
    }
}

// Returns true if the player went bust. False otherwise.
bool playerTurn(const deck_type& deck, index_type& nextCardIndex, Player& player)
{
    while (true)
    {
        if (player.score > maximumScore)
        {
            // This can happen even before the player had a choice if they drew 2
            // aces.
            std::cout << "You busted!\n";
            return true;
        }
        else
        {
            if (playerWantsHit())
            {
                int cardValue { getCardValue(deck.at(nextCardIndex++)) };
                player.score += cardValue;
                std::cout << "You were dealt a " << cardValue << " and now have " << player.score <<
'\n';
            }
            else
            {
                // The player didn't go bust.
                return false;
            }
        }
    }
}

// Returns true if the dealer went bust. False otherwise.
bool dealerTurn(const deck_type& deck, index_type& nextCardIndex, Player& dealer)
{
    // Draw cards until we reach the minimum value.
    while (dealer.score < minimumDealerScore)
    {
        int cardValue{ getCardValue(deck.at(nextCardIndex++)) };
        dealer.score += cardValue;
        dealer.score += cardValue;
        std::cout << "The dealer turned up a " << cardValue << " and now has " << dealer.score << '\n';

    }

    // If the dealer's score is too high, they went bust.
    if (dealer.score > maximumScore)
    {
        std::cout << "The dealer busted!\n";
        return true;
    }

    return false;
}

bool playBlackjack(const deck_type& deck)
{
    // Index of the card that will be drawn next. This cannot overrun
    // the array, because a player will lose before all cards are used up.
    index_type nextCardIndex{ 0 };

    // Create the dealer and give them 1 card.
    Player dealer{ getCardValue(deck.at(nextCardIndex++)) };

    // The dealer's card is face up, the player can see it.
    std::cout << "The dealer is showing: " << dealer.score << '\n';

    // Create the player and give them 2 cards.
    Player player{ getCardValue(deck.at(nextCardIndex)) + getCardValue(deck.at(nextCardIndex + 1)) };
    nextCardIndex += 2;

    std::cout << "You have: " << player.score << '\n';

    if (playerTurn(deck, nextCardIndex, player))
    {
        // The player went bust.
        return false;
    }

    if (dealerTurn(deck, nextCardIndex, dealer))
    {
        // The dealer went bust, the player wins.
        return true;
    }

    return (player.score > dealer.score);
}

int main()
{
    auto deck{ createDeck() };

    shuffleDeck(deck);

    if (playBlackjack(deck))
    {
        std::cout << "You win!\n";
    }
    else
    {
        std::cout << "You lose!\n";
    }

    return 0;
}

```

Holy moly! Where do we even begin? Don’t worry, we can do this, but we’ll need a strategy here. This Blackjack program is really composed of four parts: the logic that deals with cards, the logic that deals with the deck of cards, the logic that deals with dealing cards from the deck, and the game logic. Our strategy will be to work on each of these pieces individually, testing each part with a small test program as we go. That way, instead of trying to convert the entire program in one go, we can do it in 4 testable parts.

Start by copying the original program into your IDE, and then commenting out everything except the #include lines.

a) Let’s start by making `Card` a `class` instead of a `struct`. The good news is that the `Card` `class` is pretty similar to the `Monster` `class` from the previous quiz question. First, create private members to hold the rank and suit (name them `m_rank` and `m_suit` accordingly). Second, create a public constructor for the `Card` class so we can initialize Cards. Third, make the `class` default constructible, either by adding a default constructor or by adding default arguments to the current constructor. Fourth, because `CardSuit` and `CardRank` are tied to cards, move those into the `Card` cass as standard enums named `Suit` and `Rank`. Finally, move the `printCard()` and `getCardValue()` functions inside the `class` as public members (remember to make them `const`!).

A reminder

When using a `std::array` (or `std::vector`) where the elements are a class type, your element's`class` must have a default constructor so the elements can be initialized to a reasonable default state. If you do not provide one, you'll get a cryptic error about attempting to reference a deleted function.

The following test program should compile:

```
#include <iostream>

// ...

int main()
{
  const Card cardQueenHearts{ Card::rank_queen, Card::heart };
  cardQueenHearts.print();
  std::cout << " has the value " << cardQueenHearts.value() <<
'\n';

  return 0;
}
```

Show Solution

b) Okay, now let's work on a `Deck` `class`. The deck needs to hold 52 cards, so use a private `std::array` member to create a fixed array of 52 cards named `m_deck`. Second, create a constructor that takes no parameters and initializes m_deck with one of each card (modify the code from the original `createDeck()` function). Third, move `printDeck` into the `Deck` `class` as a public member. Fourth, move `shuffleDeck` into the class as a public member.

The trickiest part of this step is initializing the deck using the modified code from the origina`createDeck()` function. The following hint shows how to do that.

Show Hint

The following test program should compile:

```
// ...

int main()
{
  Deck deck{};

deck.print();

deck.shuffle();


deck.print();

  return 0;
```

Show Solution

c) Now we need a way to keep track of which card is next to be dealt (in the original program, this is what`nextCardIndex` was for). First, add a member named `m_cardIndex` to `Deck` and initialize it to 0. Create a public member function named `dealCard()`, which should return a const reference to the current card and advance `m_cardIndex` to the next index. `shuffle()` should also be updated to reset `m_cardIndex` (since if you shuffle the deck, you'll start dealing from the top of the deck again).

The following test program should compile:

```
// ...

int main()
{
  Deck deck{};

  deck.shuffle();
  deck.print();

  std::cout << "The first card has value: " << deck.dealCard().value() <<
'\n';
  std::cout << "The second card has value: " << deck.dealCard().value() <<
'\n';

  return 0;
}
```

Show Solution

d) Next up is the `Player`. Because `playerTurn` and `dealerTurn` are very different from each other, we'll keep them as non-member functions. Make `Player` a `class` and add a `drawCard` member function that deals the player one card from the deck, increasing the player's score. We'll also need a member function to access the `Player`'s score. For convenience, add a member function named `isBust()` that returns `true` if the player's score exceeds the maximum (`maximumScore`). The following code should compile:

```
// ...

int main()
{
    Deck deck{};

    deck.shuffle();
    deck.print();

    Player player{};
    Player dealer{};

    int playerCard { player.drawCard(deck) };
    std::cout << "The player drew a card with value " << playerCard << " and now has score " <<
player.score() << '\n';

    int dealerCard { dealer.drawCard(deck) };
    std::cout << "The dealer drew a card with value " << dealerCard << " and now has score " <<
dealer.score() << '\n';

    return 0;
}
```

Show Solution

e) Why did we write the following statement like this:

```
int playerCard { player.drawCard(deck) };
std::cout << "The player drew a card with value " << playerCard << " and now has score " <<
player.score() << '\n';
```

Instead of like this?

```
std::cout << "The player drew a card with value " << player.drawCard(deck) << " and now has score " <<
player.score() << '\n';
```

Show Solution

f) Almost there! Now, just fix up the remaining program to use the classes you wrote above. Since most of the functions have been moved into the classes, you can jettison them.

Show Solution

Next lesson

13.1 Introduction to operator overloading

Back to table of contents

Previous lesson

12.18 Timing your code

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

Name*

Email*

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

Notify me about replies: 🔔 POST COMMENT

DP N N FOUT

Newest

X