8.x — Chapter 8 comprehensive quiz

BY ALEX ON MARCH 25TH, 2016 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

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).

In C++11, non-static member initialization allows you to directly specify default values for member variables when they are declared.

Prior to C++11, constructors should not call other constructors (it will compile, but will not work as you expect). In C++11, 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 sqrt((x1 - x2)*(x1 - x2) + (y1 - y2)*(y1 - y2)). The sqrt function lives in header cmath.

The following program should run:

```
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:

```
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:

```
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;
    }

};

int main()
{
    HelloWorld hello;
```

```
30      hello.print();
31
32      return 0;
33  }
```

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.

Show Solution

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

The following program should compile:

```
1  int main()
2  {
3      Monster skele(Monster::SKELETON, "Bones", "*rattle*", 4);
4
5      return 0;
6  }
```

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 MonsterType into a std::string. Write that function (called getTypeString()), as well as a print() member function.

The following program should compile:

```
1  int main()
2  {
3      Monster skele(Monster::SKELETON, "Bones", "*rattle*", 4);
4      skele.print();
5
6      return 0;
7  }
```

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::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
    int getRandomNumber(int min, int max)
    {
        static const double fraction = 1.0 / (static_cast<double>(RAND_MAX) + 1.0);  // static
        // evenly distribute the random number across our range
        return static_cast<int>(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 MonsterType (between 0 and Monster::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 from these arrays.

The following program should compile:

```
#include <ctime> // for time()
#include <cstdlib> // for rand() and srand()
int main()
{
    srand(static_cast<unsigned int>(time(0))); // set initial seed value to system clock
    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 chapter 6 using classes! Here's the full code without classes:

```
#include <algorithm>
#include <array>
#include <cassert>
#include <ctime>
#include <iostream>
#include <random>

enum class CardSuit
{
  SUIT_CLUB,
  SUIT_DIAMOND,
  SUIT_HEART,
  SUIT_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.
```

```
52  constexpr int maximumScore{ 21 };
53
54  // Minium score that the dealer has to have.
55  constexpr int minimumDealerScore{ 17 };
56
57  void printCard(const Card& card)
58  {
59    switch (card.rank)
60    {
61    case CardRank::RANK_2:
62      std::cout << '2';
63      break;
64    case CardRank::RANK_3:
65      std::cout << '3';
66      break;
67    case CardRank::RANK_4:
68      std::cout << '4';
69      break;
70    case CardRank::RANK_5:
71      std::cout << '5';
72      break;
73    case CardRank::RANK_6:
74      std::cout << '6';
75      break;
76    case CardRank::RANK_7:
77      std::cout << '7';
78      break;
79    case CardRank::RANK_8:
80      std::cout << '8';
81      break;
82    case CardRank::RANK_9:
83      std::cout << '9';
84      break;
85    case CardRank::RANK_10:
86      std::cout << 'T';
87      break;
88    case CardRank::RANK_JACK:
89      std::cout << 'J';
90      break;
91    case CardRank::RANK_QUEEN:
92      std::cout << 'Q';
93      break;
94    case CardRank::RANK_KING:
95      std::cout << 'K';
96      break;
97    case CardRank::RANK_ACE:
98      std::cout << 'A';
99      break;
100   default:
101     std::cout << '?';
102     break;
103   }
104
105   switch (card.suit)
106   {
107   case CardSuit::SUIT_CLUB:
108     std::cout << 'C';
109     break;
110   case CardSuit::SUIT_DIAMOND:
111     std::cout << 'D';
112     break;
113   case CardSuit::SUIT_HEART:
```

```
    std::cout << 'H';
    break;
  case CardSuit::SUIT_SPADE:
    std::cout << 'S';
    break;
  default:
    std::cout << '?';
    break;
  }
}

int getCardValue(const Card& card)
{
  if (card.rank <= CardRank::RANK_10)
  {
    return (static_cast<int>(card.rank) + 2);
  }

  switch (card.rank)
  {
  case CardRank::RANK_JACK:
  case CardRank::RANK_QUEEN:
  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{};

  index_type card{ 0 };

  auto suits{ static_cast<index_type>(CardSuit::MAX_SUITS) };
  auto ranks{ static_cast<index_type>(CardRank::MAX_RANKS) };

  for (index_type suit{ 0 }; suit < suits; ++suit)
  {
    for (index_type rank{ 0 }; rank < ranks; ++rank)
    {
      deck[card].suit = static_cast<CardSuit>(suit);
      deck[card].rank = static_cast<CardRank>(rank);
      ++card;
    }
  }

```

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

    if (player.score > maximumScore)
    {
      return true;
    }
    else
    {
      if (playerWantsHit())
      {
        player.score += getCardValue(deck[nextCardIndex++]);
      }
      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)
{
  while (dealer.score < minimumDealerScore)
  {
    dealer.score += getCardValue(deck[nextCardIndex++]);
  }
```

```
238
239     return (dealer.score > maximumScore);
240   }
241
242   bool playBlackjack(const deck_type& deck)
243   {
244     index_type nextCardIndex{ 0 };
245
246     Player dealer{ getCardValue(deck[nextCardIndex++]) };
247
248     std::cout << "The dealer is showing: " << dealer.score << '\n';
249
250     Player player{ getCardValue(deck[nextCardIndex]) + getCardValue(deck[nextCardIndex + 1]) };
251     nextCardIndex += 2;
252
253     if (playerTurn(deck, nextCardIndex, player))
254     {
255       return false;
256     }
257
258     if (dealerTurn(deck, nextCardIndex, dealer))
259     {
260       return true;
261     }
262
263     return (player.score > dealer.score);
264   }
265
266   int main()
267   {
268     auto deck{ createDeck() };
269
270     shuffleDeck(deck);
271
272     if (playBlackjack(deck))
273     {
274       std::cout << "You win!\n";
275     }
276     else
277     {
278       std::cout << "You lose!\n";
279     }
280
281     return 0;
282   }
```

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. 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{ CardRank::RANK_QUEEN, CardSuit::SUIT_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 original `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:

```
// ...
```

```
2
3  int main()
4  {
5    Deck deck{};
6
7    deck.shuffle();
8    deck.print();
9
10   std::cout << "The first card has value: " << deck.dealCard().value() << '\n';
11   std::cout << "The second card has value: " << deck.dealCard().value() << '\n';
12
13   return 0;
14 }
```

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:

```
1  // ...
2
3  int main()
4  {
5    Deck deck{};
6
7    deck.shuffle();
8    deck.print();
9
10   Player player{};
11   Player dealer{};
12
13   player.drawCard(deck);
14   dealer.drawCard(deck);
15
16   std::cout << "The player drew a card with value: " << player.score() << '\n';
17   std::cout << "The dealer drew a card with value: " << dealer.score() << '\n';
18
19   return 0;
20 }
```

Show Solution

e) 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

 **9.1 -- Introduction to operator overloading**

 **Index**

8.16 -- Timing your code

C++ TUTORIAL | PRINT THIS POST

350 comments to 8.x — Chapter 8 comprehensive quiz

« Older Comments 1 ... 3 4 5

Ged
January 5, 2020 at 1:59 pm · Reply

Question number 1

Why are using static here? I tried to run the code with and without it and can't see any difference.

```
class Deck
{
private:
    std::array<Card, 52> m_deck;
    int m_cardIndex = 0;

    static int getRandomNumber(int min, int max)
    {
        static constexpr double fraction{ 1.0 / (RAND_MAX + 1.0) };
        return min + static_cast<int>((max - min + 1) * (std::rand() * fraction));
    }

    static void swapCard(Card& a, Card& b)
    {
        Card temp = a;
        a = b;
        b = temp;
    }
```

Question number 2

What is the difference between (1.0 / (RAND_MAX + 1.0)) and (1.0 / RAND_MAX)? Forgot it and just wanna make things clear.

Question number 3

When we were writing our monster code. I used std::array. Tried to use const, but it wouldn't work. So I wrote it like this. Any tips of how to improve it?

```
class MonsterGenerator
{
public:
    static std::array<std::string, 6> s_names;
    static std::array<std::string, 6> s_roars;

public:
    static int getRandomNumber(int min, int max)
    {
        static constexpr double fraction{ 1.0 / (RAND_MAX + 1.0)};
        return min + static_cast<int>((max - min + 1)* (std::rand() * fraction));
    }

    static Monster generateMonster()
    {
        return Monster(
            static_cast<Monster::MonsterType>(getRandomNumber(0,Monster::MAX_MONSTER_TYPES
            s_names[getRandomNumber(0,5)],
            s_roars[getRandomNumber(0,5)],
            getRandomNumber(1,100)
                      );
    }
};

std::array<std::string, 6> MonsterGenerator::s_names{ "Jim", "Tom", "Luke", "John", "Jacob"
std::array<std::string, 6> MonsterGenerator::s_roars{ "*rattle*", "*growl*", "*bark*", "*sc
```

nascardriver
January 7, 2020 at 2:56 am · Reply

1.
We only want to calculate `fraction` once. Without `static`, `fraction` might get calculated every time we call `getRandomNumber`. The `static` adds some runtime cost itself, I'm not sure if the `static` is actually beneficial for such a simple calculation.

2.

```
std::rand() / double(RAND_MAX) // is in [0, 1]
std::rand() / (RAND_MAX + 1.0) // is in [0, 1)
```

[0, 1] means both 0 and 1 are possible (and everything in between).
[0, 1) means 0 is possible, but 1 isn't (everything in between is possible).

If we used [0, 1], we could get

```
//                Random v
min + (max - min + 1) * 1 = max + 1
```

Oops, we got max + 1, we don't want that.
What if we remove the +1 from inside the brackets?

```
min + (max - min) * 1 = max
min + (max - min) * 0 = min
```

Seems to work, so why don't we do:

```
static int getRandomNumber(int min, int max)
```

```
{
  static constexpr double fraction{ 1.0 / (RAND_MAX)};
  return min + static_cast<int>((max - min) * (std::rand() * fraction));
}
```

Let's assume min=1, max=2
How do we generate a 2? (std::rand() * fraction) has to be exactly 1. How can that be exactly 1?
`std::rand()` has to be exactly `RAND_MAX`. In all other cases, we get a 1. That's a very predictable RNG. The RNG from the lesson doesn't have this problem. The upper half of `std::rand()`'s results generates a 2, the lower half generates a 1.

3.
`const` should work. You have to add it to the declaration AND definition.
Line 18, 19: Magic number: 5. Use `s_names.size() - 1` and `s_roars.size() - 1`.

Janko
December 19, 2019 at 6:04 am · Reply

```
std::string getTypeString() const
```

I just have a quick question, why aren't we passing (MonsterType type) to getTypeString member function here?

nascardriver
December 19, 2019 at 6:36 am · Reply

`getTypeString` is a member function, it can access `m_type` of the `Monster` it's called on.

Janko Ban
December 19, 2019 at 11:54 pm · Reply

Great, thank you! This tutorial is amazing so far, I've learned a lot.

Tompouce
November 28, 2019 at 10:31 am · Reply

Hello!
Here's a question. I was writting a function to generate a random integer with the mersene prime twister algorithm for the random monster generator. So I staticaly instantiated the mersene twister engine so that it would onle get seeded once like so:

```
static int getRandomInt(int min, int max)
{
    static std::mt19937 mersene {static_cast<std::mt19937::result_type>(std::time(nullptr))};
    static std::uniform_int_distribution RNG {min, max};

    return RNG(mersene);
}
```

BUT, as you can see I thought I could be cheeky and make the uniform_int_distribution static as well, so that it wouldn't be created every time the function is called. But then, eventhough I provided min and max values of 1 and 100 the generated numbers always stayed very low, rarely ever going beyond 8 or 9, and sometimes a 0 comes out even though I gave a minimum value of 1.

Making the distribution not static fixes everything but I'm curious as to why making it static would cause this problem

nascardriver
November 29, 2019 at 2:25 am · Reply

The first call to `getRandomInt` sets `min` and `max` of `RNG`. The `min` and `max` of every other call is ignored, because the `RNG` already exists.

```
getRandomInt(0, 8); // Set min/max to 0/8.
getRandomInt(0, 100); // Doesn't update min/max, because RNG is static.
```

Avijit Pandey
November 20, 2019 at 9:14 am · Reply

Hi! Consider this code:

```
int foo()
{
    return 5+3;
}
```

As you described in the anonymous object lesson, "return 5+3" creates an anonymous int object and assigns it the value '8' which it then returns back to the caller by value in this case.
I wanted to ask if it is possible to create a similar anonymous object in the case of the function

```
 static Monster generateMonster()
```

that is directly put in the return value?

nascardriver
November 20, 2019 at 9:23 am · Reply

The solution is creating the monster in the `return`-statement. You could remove the explicit constructor call by using brace initialization

```
return { /* ... */ };
// instead of
return Monster(/* ... */);
```

sito
October 30, 2019 at 11:10 am · Reply

hello! I've been having problems with quiz 4b. I've organized my classes in to header files with the functions in cpp files. In my deck class i'm trying to call swapCard and getRandomNumber but i get the error that the identifier can not be found. I've checked both in the header and the cpp files to make sure i didn't mess up the declaration of the function. I've been looking at the code for a while now but i can't spot what i'm doing wrong. It's probably something easy that I overlooked but it would be nice if anyone could look through my code and give me some feedback.
the error is on line 43 in the cpp file.
header file

```
#ifndef DECK_H
#define DECK_H
#include "Card.h"
class Deck {
```

```
private:
    Card m_deck[52];
    static void swapCard(Card& a, Card& b);
    static int getRandomNumber(int min, int max);

public:
    Deck();
    void printDeck();
    void shufleDeck();


};

#endif // !DECK_H
```

and the cpp file.

```
#include "Deck.h"
#include "Card.h"
#include <random>
#include <ctime>
#include <iostream>
namespace randomSeed {
    std::mt19937 mersenne(static_cast<std::mt19937::result_type>(std::time(nullptr)));

}

Deck::Deck() {
    int index{ 0 };
    for (int i{ 0 }; i < Card::RANK_MAX; ++i) {
        for (int j{ 0 }; j < Card::SUIT_MAX; ++j) {

            Deck::m_deck[index]= Card( static_cast<Card:: CardRank>(i), static_cast<Card::C
            ++index;
        }

    }

}

void Deck::printDeck() {
    for (int i{ 0 }; i < 52; ++i) {
        Card.print();
    }
}

 void Deck::swapCard(Card& a, Card& b) {
    Card temp{ a };
    a = b;
    b = temp;
}

 int Deck:: getRandomNumber(int min, int max) {
     std::uniform_int_distribution<>number(min, max);
     return number(randomSeed::mersenne);
 }

 void shufleDeck() {
     for (int i{ 0 }; i < 52; ++i) {
         int number{ getRandomNumber(0, 51) };
         swapCard(Deck::m_deck[i], Deck::m_deck[number]);
     }

```

```
47  }
```

nascardriver
October 31, 2019 at 4:09 am · Reply

Hi!

`sufleDeck` the the cpp file isn't a member of `Deck`. You forgot the `Deck::` prefix.

```
1  void shufleDeck()
2  void Deck::shufleDeck()
```

Samira Ferdi
October 10, 2019 at 6:43 pm · Reply

Hi, Alex and Nascardriver!

I try to move the MonsterGenerator class to header file or .cpp file. The error is same: "MonsterGenerator has not been declared". I don't know why. But, I think "Is it because the static keyword 'means' the MonsterGenerator is only seen in that header file or .cpp file?" If it is, so, there is limitation of static class in term of multiple files. But, I'm not sure about this.

My second question is why don't make s_names and s_roars private? I try to do that and errors come in:

```
1  ||=== Build: Debug in Chapter8_quiz (compiler: GNU GCC Compiler) ===|
2  D:\tugas-wiwid\c++\Chapter8_quiz\main.cpp|12|error: in-class initialization of static data m
3  D:\tugas-wiwid\c++\Chapter8_quiz\main.cpp|12|error: non-constant in-class initialization inv
4  D:\tugas-wiwid\c++\Chapter8_quiz\main.cpp|12|note: (an out of class initialization is requir
5  D:\tugas-wiwid\c++\Chapter8_quiz\main.cpp|15|error: in-class initialization of static data m
6  D:\tugas-wiwid\c++\Chapter8_quiz\main.cpp|15|error: non-constant in-class initialization inv
7  ||=== Build failed: 4 error(s), 0 warning(s) (0 minute(s), 1 second(s)) ===|
```

Why an out of class initialization is still required? s_names and s_roars are static const member variable.

nascardriver
October 11, 2019 at 1:11 am · Reply

`MonsterGenerator` can be moved into separate files. The error message looks like you missed some includes. Without code and a full error message, I can't help.

> why don't make s_names and s_roars private?
They're not used outside of `generateMonster`, so there's no reason to make them accessible there.
`static` member variables that aren't literals have to be initialized outside of the class, as otherwise it's not clear when to initialize them.

```
1  class MonsterGenerator
2  {
3  private:
4    const std::string s_names[6];
5    static const std::string s_roars[6];
6    // ...
7  };
8
9  const std::string MonsterGenerator::s_names[6]{ "Blarg", "Moog", "Pksh", "Tyrn", "Mor
10 const std::string MonsterGenerator::s_roars[6]{ "*ROAR*", "*peep*", "*squeal*", "*whi
```

Alternatively, they can be marked `inline`. They will be initialized when they're first used.

```
class MonsterGenerator
{
private:
  static inline const std::string s_names[6]{ "Blarg", "Moog", "Pksh", "Tyrn", "Mort",
  static inline const std::string s_roars[6]{ "*ROAR*", "*peep*", "*squeal*", "*whine*
  // ...
```

This is easier, but adds overhead to the run time as every time one of the members is accessed, it has to be checked if they have been initialized already.

Samira Ferdi
October 9, 2019 at 5:23 pm · Reply

Hi, Alex and Nascardriver!

In quiz no.2, my compiler throw errors that I don't understand why:

```
||=== Build: Debug in Chapter8_quiz (compiler: GNU GCC Compiler) ===|
D:\tugas-wiwid\c++\Chapter8_quiz\main.cpp|4|error: 'class HelloWorld' has pointer data membe
D:\tugas-wiwid\c++\Chapter8_quiz\main.cpp|4|error:   but does not override 'HelloWorld(const
D:\tugas-wiwid\c++\Chapter8_quiz\main.cpp|4|error:   or 'operator=(const HelloWorld&)' [-Wer
D:\tugas-wiwid\c++\Chapter8_quiz\main.cpp||In constructor 'HelloWorld::HelloWorld()':|
D:\tugas-wiwid\c++\Chapter8_quiz\main.cpp|10|error: 'HelloWorld::m_data' should be initializ
||=== Build failed: 4 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===|
```

nascardriver
October 10, 2019 at 6:11 am · Reply

Hi!

`m_data` is a pointer. When you create a copy of the class, the pointer will be copied.

```
{
  HelloWorld hello{};
  HelloWorld hi{ hello }; // Create a copy of @hello

  // @hello.m_data and @hi.m_data are the same.
  // They both go out of scope now and delete their @m_data.

  // @hi deletes its @m_data. @hello.m_data pointed to the same
  // memory location. Its pointer is now dangling!

  // @hello tries to delete its @m_data. It has already been
  // deleted by @hi -> Undefined behavior.
}
```

The same happens when you assign one `HelloWorld` to another.
The copy that only copies the pointer, but not the pointed-to data is called _shallow copy_. It's covered later.

You can explicitly default the copy constructor and copy assignment operator to silence the warnings.

```
// Add these lines to the public section of @HelloWorld
HelloWorld(const HelloWorld&) = default;
HelloWorld& operator=(const HelloWorld&) = default; // Covered later
```

« Older Comments 1 ... 3 4 5