8.x — Chapter 8 comprehensive quiz

BY ALEX ON MARCH 25TH, 2016 | LAST MODIFIED BY ALEX ON JUNE 29TH, 2018

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

Chapter 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

1a) 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:

```cpp
#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);
```

Hide Solution

```cpp
#include <iostream>

class Point2d
{
private:
    double m_x;
    double m_y;

public:
    Point2d(double x = 0.0, double y = 0.0)
        : m_x(x), m_y(y)
    {
    }

    void print() const
    {
        std::cout << "Point2d(" << m_x << ", " << m_y << ")\n";
    }
};


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

    return 0;
}
```

1b) 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:

```cpp
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
```

Hide Solution

```cpp
#include <cmath>
#include <iostream>

class Point2d
{
private:
    double m_x;
    double m_y;

public:
    Point2d(double x = 0.0, double y = 0.0)
        : m_x(x), m_y(y)
    {
    }

    void print() const
    {
        std::cout << "Point2d(" << m_x << " , " << m_y << ")\n";
    }

    double distanceTo(const Point2d & other) const
    {
        return sqrt((m_x - other.m_x)*(m_x - other.m_x) + (m_y - other.m_y)*(m_y - other.m_y));
    }
};

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;
}
```

1c) 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:

```cpp
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
```

Hide Solution

```cpp
#include <cmath>
#include <iostream>

class Point2d
{
private:
    double m_x;
    double m_y;

public:
    Point2d(double x = 0.0, double y = 0.0)
        : m_x(x), m_y(y)
    {
    }

    void print() const
    {
        std::cout << "Point2d(" << m_x << " , " << m_y << ")\n";
    }

    friend double distanceFrom(const Point2d &x, const Point2d &y);

};

double distanceFrom(const Point2d &x, const Point2d &y)
{
    return sqrt((x.m_x - y.m_x)*(x.m_x - y.m_x) + (x.m_y - y.m_y)*(x.m_y - y.m_y));
}

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;
}
```

2) Write a destructor for this class:

```cpp
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;
    hello.print();

    return 0;
}
```

Hide Solution

```
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()
    {
        delete[] m_data;
    }

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

};

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

    return 0;
}
```

3) Let's create a random monster generator. This one should be fun.

3a) 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.

Hide Solution

```
enum MonsterType
{
    DRAGON,
    GOBLIN,
    OGRE,
    ORC,
    SKELETON,
    TROLL,
    VAMPIRE,
    ZOMBIE,
```

```
    MAX_MONSTER_TYPES
};
```

3b) 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.

Hide Solution

```
#include <string>

enum MonsterType
{
    DRAGON,
    GOBLIN,
    OGRE,
    ORC,
    SKELETON,
    TROLL,
    VAMPIRE,
    ZOMBIE,
    MAX_MONSTER_TYPES
};

class Monster
{
private:

    MonsterType m_type;
    std::string m_name;
    std::string m_roar;
    int m_hitPoints;
};
```

3c) enum MonsterType is specific to Monster, so move the enum inside the class as a public declaration.

Hide Solution

```
#include <string>

class Monster
{
public:
    enum MonsterType
    {
        DRAGON,
        GOBLIN,
        OGRE,
        ORC,
        SKELETON,
        TROLL,
        VAMPIRE,
        ZOMBIE,
        MAX_MONSTER_TYPES
    };

private:

    MonsterType m_type;
    std::string m_name;
    std::string m_roar;
    int m_hitPoints;
};
```

3d) Create a constructor that allows you to initialize all of the member variables.

The following program should compile:

```
int main()
```

```
{
    Monster skele(Monster::SKELETON, "Bones", "*rattle*", 4);

    return 0;
}
```

Hide Solution

```
#include <string>

class Monster
{
public:
    enum MonsterType
    {
        DRAGON,
        GOBLIN,
        OGRE,
        ORC,
        SKELETON,
        TROLL,
        VAMPIRE,
        ZOMBIE,
        MAX_MONSTER_TYPES
    };

private:

    MonsterType m_type;
    std::string m_name;
    std::string m_roar;
    int m_hitPoints;

public:
    Monster(MonsterType type, std::string name, std::string roar, int hitPoints)
        : m_type(type), m_name(name), m_roar(roar), m_hitPoints(hitPoints)
    {

    }
};

int main()
{
    Monster skele(Monster::SKELETON, "Bones", "*rattle*", 4);

    return 0;
}
```

3e) 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:

```
int main()
{
    Monster skele(Monster::SKELETON, "Bones", "*rattle*", 4);
    skele.print();

    return 0;
}
```

and print:

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

Hide Solution

```cpp
#include <iostream>
#include <string>

class Monster
{
public:
    enum MonsterType
    {
        DRAGON,
        GOBLIN,
        OGRE,
        ORC,
        SKELETON,
        TROLL,
        VAMPIRE,
        ZOMBIE,
        MAX_MONSTER_TYPES
    };

private:

    MonsterType m_type;
    std::string m_name;
    std::string m_roar;
    int m_hitPoints;

public:
    Monster(MonsterType type, std::string name, std::string roar, int hitPoints)
        : m_type(type), m_name(name), m_roar(roar), m_hitPoints(hitPoints)
    {

    }

    std::string getTypeString() const
    {
        switch (m_type)
        {
        case DRAGON: return "dragon";
        case GOBLIN: return "goblin";
        case OGRE: return "ogre";
        case ORC: return "orc";
        case SKELETON: return "skeleton";
        case TROLL: return "troll";
        case VAMPIRE: return "vampire";
        case ZOMBIE: return "zombie";
        }

        return "???";
    }

    void print() const
    {
        std::cout << m_name << " the " << getTypeString() << " has " << m_hitPoints << " hit points and
 says " << m_roar << '\n';
    }
};

int main()
{
    Monster skele(Monster::SKELETON, "Bones", "*rattle*", 4);
    skele.print();

    return 0;
}
```

3f) 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*
```

Hide Solution

```
#include <iostream>
#include <string>

class Monster
{
public:
    enum MonsterType
    {
        DRAGON,
        GOBLIN,
        OGRE,
        ORC,
        SKELETON,
        TROLL,
        VAMPIRE,
        ZOMBIE,
        MAX_MONSTER_TYPES
    };

private:

    MonsterType m_type;
    std::string m_name;
    std::string m_roar;
    int m_hitPoints;

public:
    Monster(MonsterType type, std::string name, std::string roar, int hitPoints)
        : m_type(type), m_name(name), m_roar(roar), m_hitPoints(hitPoints)
    {

    }

    std::string getTypeString() const
    {
        switch (m_type)
        {
        case DRAGON: return "dragon";
        case GOBLIN: return "goblin";
        case OGRE: return "ogre";
        case ORC: return "orc";
        case SKELETON: return "skeleton";
        case TROLL: return "troll";
        case VAMPIRE: return "vampire";
        case ZOMBIE: return "zombie";
        }
```

```
        return "???";
    }

    void print() const
    {
        std::cout << m_name << " the " << getTypeString() << " has " << m_hitPoints << " hit points and
 says " << m_roar << '\n';
    }
};

class MonsterGenerator
{
public:
    static Monster generateMonster()
    {
        return Monster(Monster::SKELETON, "Bones", "*rattle*", 4);
    }
};

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

    return 0;
}
```

3g) 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 used for
efficiency, so we only calculate this value once
        // 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.

Hide Solution

```
class MonsterGenerator
{
public:
    // Generate a random number between min and max (inclusive)
    // Assumes srand() has already been called
    static int getRandomNumber(int min, int max)
    {
        static const 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>(rand() * fraction * (max - min + 1) + min);
    }

    static Monster generateMonster()
    {
        return Monster(Monster::SKELETON, "Bones", "*rattle*", 4);
    }
};
```

3h) 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;
}
```

Hide Solution

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

class Monster
{
public:
    enum MonsterType
    {
        DRAGON,
        GOBLIN,
        OGRE,
        ORC,
        SKELETON,
        TROLL,
        VAMPIRE,
        ZOMBIE,
        MAX_MONSTER_TYPES
    };

private:

    MonsterType m_type;
    std::string m_name;
    std::string m_roar;
    int m_hitPoints;

public:
    Monster(MonsterType type, std::string name, std::string roar, int hitPoints)
        : m_type(type), m_name(name), m_roar(roar), m_hitPoints(hitPoints)
    {

    }

    std::string getTypeString() const
    {
        switch (m_type)
        {
        case DRAGON: return "dragon";
        case GOBLIN: return "goblin";
        case OGRE: return "ogre";
        case ORC: return "orc";
        case SKELETON: return "skeleton";
        case TROLL: return "troll";
        case VAMPIRE: return "vampire";
        case ZOMBIE: return "zombie";
        }

        return "???";
```

```
    }

    void print() const
    {
        std::cout << m_name << " the " << getTypeString() << " has " << m_hitPoints << " hit points and
 says " << m_roar << '\n';
    }
};

class MonsterGenerator
{
public:
    // Generate a random number between min and max (inclusive)
    // Assumes srand() has already been called
    static int getRandomNumber(int min, int max)
    {
        static const 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>(rand() * fraction * (max - min + 1) + min);
    }

    static Monster generateMonster()
    {
        Monster::MonsterType type = static_cast<Monster::MonsterType>(getRandomNumber(0, Monster::MAX_M
ONSTER_TYPES - 1));
        int hitPoints = getRandomNumber(1, 100);

        static std::string s_names[6]{ "Blarg", "Moog", "Pksh", "Tyrn", "Mort", "Hans" };
        static std::string s_roars[6]{ "*ROAR*", "*peep*", "*squeal*", "*whine*", "*hum*", "*burp*"};

        return Monster(type, s_names[getRandomNumber(0, 5)], s_roars[getRandomNumber(0, 5)], hitPoints)
;
    }
};

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;
}
```

3i) Why did we declare variables s_names and s_roars as static?

Hide Solution

Making s_names and s_roars static causes them to be initialized only once. Otherwise, they would get reinitialized every time generateMonster() was called.

4) Okay, time for that game face again. This one is going to be a challenge. Let's rewrite the Blackjack games we wrote in chapter 6 using classes! Here's the full code without classes:

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

enum CardSuit
{
    SUIT_CLUB,
    SUIT_DIAMOND,
    SUIT_HEART,
```

```
    SUIT_SPADE,
    MAX_SUITS
};

enum 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;
};

void printCard(const Card &card)
{
    switch (card.rank)
    {
        case RANK_2:        std::cout << '2'; break;
        case RANK_3:        std::cout << '3'; break;
        case RANK_4:        std::cout << '4'; break;
        case RANK_5:        std::cout << '5'; break;
        case RANK_6:        std::cout << '6'; break;
        case RANK_7:        std::cout << '7'; break;
        case RANK_8:        std::cout << '8'; break;
        case RANK_9:        std::cout << '9'; break;
        case RANK_10:       std::cout << 'T'; break;
        case RANK_JACK:     std::cout << 'J'; break;
        case RANK_QUEEN:    std::cout << 'Q'; break;
        case RANK_KING:     std::cout << 'K'; break;
        case RANK_ACE:      std::cout << 'A'; break;
    }

    switch (card.suit)
    {
        case SUIT_CLUB:     std::cout << 'C'; break;
        case SUIT_DIAMOND:  std::cout << 'D'; break;
        case SUIT_HEART:    std::cout << 'H'; break;
        case SUIT_SPADE:    std::cout << 'S'; break;
    }
}

void printDeck(const std::array<Card, 52> deck)
{
    for (const auto &card : deck)
    {
        printCard(card);
        std::cout << ' ';
    }

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

```
78  void swapCard(Card &a, Card &b)
79  {
80      Card temp = a;
81      a = b;
82      b = temp;
83  }
84
85  // Generate a random number between min and max (inclusive)
86  // Assumes srand() has already been called
87  int getRandomNumber(int min, int max)
88  {
89      static const double fraction = 1.0 / (static_cast<double>(RAND_MAX) + 1.0);  // static used for ef
90  ficiency, so we only calculate this value once
91      // evenly distribute the random number across our range
92      return static_cast<int>(rand() * fraction * (max - min + 1) + min);
93  }
94
95  void shuffleDeck(std::array<Card, 52> &deck)
96  {
97      // Step through each card in the deck
98      for (int index = 0; index < 52; ++index)
99      {
100         // Pick a random card, any card
101         int swapIndex = getRandomNumber(0, 51);
102         // Swap it with the current card
103         swapCard(deck[index], deck[swapIndex]);
104     }
105 }
106
107 int getCardValue(const Card &card)
108 {
109     switch (card.rank)
110     {
111     case RANK_2:         return 2;
112     case RANK_3:         return 3;
113     case RANK_4:         return 4;
114     case RANK_5:         return 5;
115     case RANK_6:         return 6;
116     case RANK_7:         return 7;
117     case RANK_8:         return 8;
118     case RANK_9:         return 9;
119     case RANK_10:        return 10;
120     case RANK_JACK:      return 10;
121     case RANK_QUEEN:     return 10;
122     case RANK_KING:      return 10;
123     case RANK_ACE:       return 11;
124     }
125
126     return 0;
127 }
128
129 char getPlayerChoice()
130 {
131     std::cout << "(h) to hit, or (s) to stand: ";
132     char choice;
133     do
134     {
135         std::cin >> choice;
136     } while (choice != 'h' && choice != 's');
137
138     return choice;
139 }
140
141 bool playBlackjack(const std::array<Card, 52> deck)
142 {
143     // Set up the initial game state
144     const Card *cardPtr = &deck[0];
```

```
145
146     int playerTotal = 0;
147     int dealerTotal = 0;
148
149     // Deal the dealer one card
150     dealerTotal += getCardValue(*cardPtr++);
151     std::cout << "The dealer is showing: " << dealerTotal << '\n';
152
153     // Deal the player two cards
154     playerTotal += getCardValue(*cardPtr++);
155     playerTotal += getCardValue(*cardPtr++);
156
157     // Player goes first
158     while (1)
159     {
160         std::cout << "You have: " << playerTotal << '\n';
161         char choice = getPlayerChoice();
162         if (choice == 's')
163             break;
164
165         playerTotal += getCardValue(*cardPtr++);
166
167         // See if the player busted
168         if (playerTotal > 21)
169             return false;
170     }
171
172     // If player hasn't busted, dealer goes until he has at least 17 points
173     while (dealerTotal < 17)
174     {
175         dealerTotal += getCardValue(*cardPtr++);
176         std::cout << "The dealer now has: " << dealerTotal << '\n';
177     }
178
179     // If dealer busted, player wins
180     if (dealerTotal > 21)
181         return true;
182
183     return (playerTotal > dealerTotal);
184 }
185
186 int main()
187 {
188     srand(static_cast<unsigned int>(time(0))); // set initial seed value to system clock
189     rand(); // If using Visual Studio, discard first random value
190
191     std::array<Card, 52> deck;
192
193     // We could initialize each card individually, but that would be a pain.  Let's use a loop.
194     int card = 0;
195     for (int suit = 0; suit < MAX_SUITS; ++suit)
196     for (int rank = 0; rank < MAX_RANKS; ++rank)
197     {
198         deck[card].suit = static_cast<CardSuit>(suit);
199         deck[card].rank = static_cast<CardRank>(rank);
200         ++card;
201     }
202
203     shuffleDeck(deck);
204
205     if (playBlackjack(deck))
206         std::cout << "You win!\n";
207     else
208         std::cout << "You lose!\n";
209
210     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.

4a) 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, move the enums for CardSuit, CardRank inside the card class as public definitions (they're intrinsically related to Card, so it makes more sense for them to be inside the class, not outside). Second, create private members to hold the CardRank and CardSuit (name them m_rank and m_suit accordingly). Third, create a public constructor for the Card class so we can initialize Cards. Forth, make sure to assign default values to the parameters so this can be used as a default constructor (pick any values you like). Finally, move the printCard() and getCardValue() functions inside the class as public members (remember to make them const!).

Important note: 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::SUIT_HEART);
    cardQueenHearts.printCard();
    std::cout << " has the value " << cardQueenHearts.getCardValue() << '\n';

    return 0;
}
```

Hide Solution

```
#include <iostream>

class Card
{
public:
    enum CardSuit
    {
        SUIT_CLUB,
        SUIT_DIAMOND,
        SUIT_HEART,
        SUIT_SPADE,
        MAX_SUITS
    };

    enum 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
    };

```

```
private:
    CardRank m_rank;
    CardSuit m_suit;

public:
    Card(CardRank rank=MAX_RANKS, CardSuit suit=MAX_SUITS) :
        m_rank(rank), m_suit(suit)
    {

    }

    void printCard() const
    {
        switch (m_rank)
        {
        case RANK_2:        std::cout << '2'; break;
        case RANK_3:        std::cout << '3'; break;
        case RANK_4:        std::cout << '4'; break;
        case RANK_5:        std::cout << '5'; break;
        case RANK_6:        std::cout << '6'; break;
        case RANK_7:        std::cout << '7'; break;
        case RANK_8:        std::cout << '8'; break;
        case RANK_9:        std::cout << '9'; break;
        case RANK_10:       std::cout << 'T'; break;
        case RANK_JACK:     std::cout << 'J'; break;
        case RANK_QUEEN:    std::cout << 'Q'; break;
        case RANK_KING:     std::cout << 'K'; break;
        case RANK_ACE:      std::cout << 'A'; break;
        }

        switch (m_suit)
        {
        case SUIT_CLUB:     std::cout << 'C'; break;
        case SUIT_DIAMOND:  std::cout << 'D'; break;
        case SUIT_HEART:    std::cout << 'H'; break;
        case SUIT_SPADE:    std::cout << 'S'; break;
        }
    }

    int getCardValue() const
    {
        switch (m_rank)
        {
        case RANK_2:        return 2;
        case RANK_3:        return 3;
        case RANK_4:        return 4;
        case RANK_5:        return 5;
        case RANK_6:        return 6;
        case RANK_7:        return 7;
        case RANK_8:        return 8;
        case RANK_9:        return 9;
        case RANK_10:       return 10;
        case RANK_JACK:     return 10;
        case RANK_QUEEN:    return 10;
        case RANK_KING:     return 10;
        case RANK_ACE:      return 11;
        }

        return 0;
    }
};

int main()
{
    const Card cardQueenHearts(Card::RANK_QUEEN, Card::SUIT_HEART);
    cardQueenHearts.printCard();
    std::cout << " has the value " << cardQueenHearts.getCardValue() << '\n';
```

```
    return 0;
}
```

4b) 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 main() function). Inside the initialization loop, create an anonymous Card object and assign it to your deck element. Third, move printDeck into the Deck class as a public member. Fourth, move getRandomNumber() and swapCard() into the Deck class as a private static members (they're just helper functions, so they don't need access to *this). Fifth, move shuffleDeck into the class as a public member.

Hint: The trickiest part of this step is initializing the deck using the modified code from the original main() function. The following line shows how to do that.

```
    m_deck[card] = Card(static_cast<Card::CardRank>(rank), static_cast<Card::CardSuit>(suit));
```

The following test program should compile:

```
#include <iostream>
#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

    Deck deck;
    deck.printDeck();
    deck.shuffleDeck();
    deck.printDeck();

    return 0;
}
```

Hide Solution

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

class Card
{
public:
    enum CardSuit
    {
        SUIT_CLUB,
        SUIT_DIAMOND,
        SUIT_HEART,
        SUIT_SPADE,
        MAX_SUITS
    };

    enum 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
    };

private:
    CardRank m_rank;
    CardSuit m_suit;

public:

    Card(CardRank rank=MAX_RANKS, CardSuit suit=MAX_SUITS) :
        m_rank(rank), m_suit(suit)
    {

    }

    void printCard() const
    {
        switch (m_rank)
        {
        case RANK_2:        std::cout << '2'; break;
        case RANK_3:        std::cout << '3'; break;
        case RANK_4:        std::cout << '4'; break;
        case RANK_5:        std::cout << '5'; break;
        case RANK_6:        std::cout << '6'; break;
        case RANK_7:        std::cout << '7'; break;
        case RANK_8:        std::cout << '8'; break;
        case RANK_9:        std::cout << '9'; break;
        case RANK_10:       std::cout << 'T'; break;
        case RANK_JACK:     std::cout << 'J'; break;
        case RANK_QUEEN:    std::cout << 'Q'; break;
        case RANK_KING:     std::cout << 'K'; break;
        case RANK_ACE:      std::cout << 'A'; break;
        }

        switch (m_suit)
        {
        case SUIT_CLUB:     std::cout << 'C'; break;
        case SUIT_DIAMOND:  std::cout << 'D'; break;
        case SUIT_HEART:    std::cout << 'H'; break;
        case SUIT_SPADE:    std::cout << 'S'; break;
        }
    }

    int getCardValue() const
    {
        switch (m_rank)
        {
        case RANK_2:        return 2;
        case RANK_3:        return 3;
        case RANK_4:        return 4;
        case RANK_5:        return 5;
        case RANK_6:        return 6;
        case RANK_7:        return 7;
        case RANK_8:        return 8;
        case RANK_9:        return 9;
        case RANK_10:       return 10;
        case RANK_JACK:     return 10;
        case RANK_QUEEN:    return 10;
        case RANK_KING:     return 10;
        case RANK_ACE:      return 11;
        }

        return 0;
    }
};
```

```
98
99  class Deck
100 {
101 private:
102     std::array<Card, 52> m_deck;
103
104     // Generate a random number between min and max (inclusive)
105     // Assumes srand() has already been called
106     static int getRandomNumber(int min, int max)
107     {
108         static const double fraction = 1.0 / (static_cast<double>(RAND_MAX) + 1.0);  // static used fo
109 r efficiency, so we only calculate this value once
110         // evenly distribute the random number across our range
111         return static_cast<int>(rand() * fraction * (max - min + 1) + min);
112     }
113
114     static void swapCard(Card &a, Card &b)
115     {
116         Card temp = a;
117         a = b;
118         b = temp;
119     }
120
121 public:
122     Deck()
123     {
124         int card = 0;
125         for (int suit = 0; suit < Card::MAX_SUITS; ++suit)
126             for (int rank = 0; rank < Card::MAX_RANKS; ++rank)
127             {
128                 m_deck[card] = Card(static_cast<Card::CardRank>(rank), static_cast<Card::CardSuit>(sui
129 t));
130                 ++card;
131             }
132     }
133
134     void printDeck() const
135     {
136         for (const auto &card : m_deck)
137         {
138             card.printCard();
139             std::cout << ' ';
140         }
141
142         std::cout << '\n';
143     }
144
145
146     void shuffleDeck()
147     {
148         // Step through each card in the deck
149         for (int index = 0; index < 52; ++index)
150         {
151             // Pick a random card, any card
152             int swapIndex = getRandomNumber(0, 51);
153             // Swap it with the current card
154             swapCard(m_deck[index], m_deck[swapIndex]);
155         }
156     }
157 };
158
159 int main()
160 {
161     srand(static_cast<unsigned int>(time(0))); // set initial seed value to system clock
162     rand(); // If using Visual Studio, discard first random value
163
164     Deck deck;
```

```
    deck.printDeck();
    deck.shuffleDeck();
    deck.printDeck();

    return 0;
}
```

4c) Now we need a way to keep track of which card is next to be dealt (in the original program, this is what cardptr was for). First, add a int member named m_cardIndex and initialize it to 0. Create a public member function named dealCard(), which should return a const reference to the current card and advance the index. shuffleDeck() 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()
{
    srand(static_cast<unsigned int>(time(0))); // set initial seed value to system clock
    rand(); // If using Visual Studio, discard first random value

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

    return 0;
}
```

Hide Solution

```
#include <iostream>
#include <array>
#include <ctime> // for time()
#include <cstdlib> // for rand() and srand()
#include <cassert> // for assert()

class Card
{
public:
    enum CardSuit
    {
        SUIT_CLUB,
        SUIT_DIAMOND,
        SUIT_HEART,
        SUIT_SPADE,
        MAX_SUITS
    };

    enum 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
    };

private:
```

```
    CardRank m_rank;
    CardSuit m_suit;

public:

    Card(CardRank rank=MAX_RANKS, CardSuit suit=MAX_SUITS) :
        m_rank(rank), m_suit(suit)
    {

    }

    void printCard() const
    {
        switch (m_rank)
        {
        case RANK_2:        std::cout << '2'; break;
        case RANK_3:        std::cout << '3'; break;
        case RANK_4:        std::cout << '4'; break;
        case RANK_5:        std::cout << '5'; break;
        case RANK_6:        std::cout << '6'; break;
        case RANK_7:        std::cout << '7'; break;
        case RANK_8:        std::cout << '8'; break;
        case RANK_9:        std::cout << '9'; break;
        case RANK_10:       std::cout << 'T'; break;
        case RANK_JACK:     std::cout << 'J'; break;
        case RANK_QUEEN:    std::cout << 'Q'; break;
        case RANK_KING:     std::cout << 'K'; break;
        case RANK_ACE:      std::cout << 'A'; break;
        }

        switch (m_suit)
        {
        case SUIT_CLUB:     std::cout << 'C'; break;
        case SUIT_DIAMOND:  std::cout << 'D'; break;
        case SUIT_HEART:    std::cout << 'H'; break;
        case SUIT_SPADE:    std::cout << 'S'; break;
        }
    }

    int getCardValue() const
    {
        switch (m_rank)
        {
        case RANK_2:        return 2;
        case RANK_3:        return 3;
        case RANK_4:        return 4;
        case RANK_5:        return 5;
        case RANK_6:        return 6;
        case RANK_7:        return 7;
        case RANK_8:        return 8;
        case RANK_9:        return 9;
        case RANK_10:       return 10;
        case RANK_JACK:     return 10;
        case RANK_QUEEN:    return 10;
        case RANK_KING:     return 10;
        case RANK_ACE:      return 11;
        }

        return 0;
    }
};

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

```
105
106     // Generate a random number between min and max (inclusive)
107     // Assumes srand() has already been called
108     static int getRandomNumber(int min, int max)
109     {
110         static const double fraction = 1.0 / (static_cast<double>(RAND_MAX) + 1.0);  // static used fo
111 r efficiency, so we only calculate this value once
112         // evenly distribute the random number across our range
113         return static_cast<int>(rand() * fraction * (max - min + 1) + min);
114     }
115
116     static void swapCard(Card &a, Card &b)
117     {
118         Card temp = a;
119         a = b;
120         b = temp;
121     }
122
123 public:
124     Deck()
125     {
126         int card = 0;
127         for (int suit = 0; suit < Card::MAX_SUITS; ++suit)
128             for (int rank = 0; rank < Card::MAX_RANKS; ++rank)
129             {
130                 m_deck[card] = Card(static_cast<Card::CardRank>(rank), static_cast<Card::CardSuit>(sui
131 t));
132                 ++card;
133             }
134     }
135
136     void printDeck() const
137     {
138         for (const auto &card : m_deck)
139         {
140             card.printCard();
141             std::cout << ' ';
142         }
143
144         std::cout << '\n';
145     }
146
147
148     void shuffleDeck()
149     {
150         // Step through each card in the deck
151         for (int index = 0; index < 52; ++index)
152         {
153             // Pick a random card, any card
154             int swapIndex = getRandomNumber(0, 51);
155             // Swap it with the current card
156             swapCard(m_deck[index], m_deck[swapIndex]);
157         }
158
159     m_cardIndex = 0; // start a new deal
160     }
161
162     const Card& dealCard()
163     {
164         assert (m_cardIndex < 52);
165         return m_deck[m_cardIndex++];
166     }
167 };
168
169 int main()
170 {
171     srand(static_cast<unsigned int>(time(0))); // set initial seed value to system clock
```

```
    rand(); // If using Visual Studio, discard first random value

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

    return 0;
}
```

4d) Almost there! Now, just fix up the remaining program to use the classes you wrote above. Since most of the initialization routines has been moved into the classes, you can jettison them.

Hide Solution

```
#include <iostream>
#include <array>
#include <ctime> // for time()
#include <cstdlib> // for rand() and srand()
#include <cassert> // for assert()

class Card
{
public:
    enum CardSuit
    {
        SUIT_CLUB,
        SUIT_DIAMOND,
        SUIT_HEART,
        SUIT_SPADE,
        MAX_SUITS
    };

    enum 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
    };

private:
    CardRank m_rank;
    CardSuit m_suit;

public:

    Card(CardRank rank=MAX_RANKS, CardSuit suit=MAX_SUITS) :
        m_rank(rank), m_suit(suit)
    {

    }

    void printCard() const
    {
        switch (m_rank)
```

```
        {
        case RANK_2:        std::cout << '2'; break;
        case RANK_3:        std::cout << '3'; break;
        case RANK_4:        std::cout << '4'; break;
        case RANK_5:        std::cout << '5'; break;
        case RANK_6:        std::cout << '6'; break;
        case RANK_7:        std::cout << '7'; break;
        case RANK_8:        std::cout << '8'; break;
        case RANK_9:        std::cout << '9'; break;
        case RANK_10:       std::cout << 'T'; break;
        case RANK_JACK:     std::cout << 'J'; break;
        case RANK_QUEEN:    std::cout << 'Q'; break;
        case RANK_KING:     std::cout << 'K'; break;
        case RANK_ACE:      std::cout << 'A'; break;
        }

        switch (m_suit)
        {
        case SUIT_CLUB:     std::cout << 'C'; break;
        case SUIT_DIAMOND:  std::cout << 'D'; break;
        case SUIT_HEART:    std::cout << 'H'; break;
        case SUIT_SPADE:    std::cout << 'S'; break;
        }
    }

    int getCardValue() const
    {
        switch (m_rank)
        {
        case RANK_2:        return 2;
        case RANK_3:        return 3;
        case RANK_4:        return 4;
        case RANK_5:        return 5;
        case RANK_6:        return 6;
        case RANK_7:        return 7;
        case RANK_8:        return 8;
        case RANK_9:        return 9;
        case RANK_10:       return 10;
        case RANK_JACK:     return 10;
        case RANK_QUEEN:    return 10;
        case RANK_KING:     return 10;
        case RANK_ACE:      return 11;
        }

        return 0;
    }
};

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

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

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

```
        a = b;
        b = temp;
    }

public:
    Deck()
    {
        int card = 0;
        for (int suit = 0; suit < Card::MAX_SUITS; ++suit)
            for (int rank = 0; rank < Card::MAX_RANKS; ++rank)
            {
                m_deck[card] = Card(static_cast<Card::CardRank>(rank), static_cast<Card::CardSuit>(sui
t));
                ++card;
            }
    }

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

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


    void shuffleDeck()
    {
        // Step through each card in the deck
        for (int index = 0; index < 52; ++index)
        {
            // Pick a random card, any card
            int swapIndex = getRandomNumber(0, 51);
            // Swap it with the current card
            swapCard(m_deck[index], m_deck[swapIndex]);
        }

    m_cardIndex = 0; // start a new deal
    }

    const Card& dealCard()
    {
        assert (m_cardIndex < 52);
        return m_deck[m_cardIndex++];
    }
};


char getPlayerChoice()
{
    std::cout << "(h) to hit, or (s) to stand: ";
    char choice;
    do
    {
        std::cin >> choice;
    } while (choice != 'h' && choice != 's');

    return choice;
}

bool playBlackjack(Deck &deck)
{

    int playerTotal = 0;
```

```
    int dealerTotal = 0;

    // Deal the dealer one card
    dealerTotal += deck.dealCard().getCardValue();
    std::cout << "The dealer is showing: " << dealerTotal << '\n';

    // Deal the player two cards
    playerTotal += deck.dealCard().getCardValue();
    playerTotal += deck.dealCard().getCardValue();

    // Player goes first
    while (1)
    {
        std::cout << "You have: " << playerTotal << '\n';
        char choice = getPlayerChoice();
        if (choice == 's')
            break;

        playerTotal += deck.dealCard().getCardValue();

        // See if the player busted
        if (playerTotal > 21)
            return false;
    }

    // If player hasn't busted, dealer goes until he has at least 17 points
    while (dealerTotal < 17)
    {
        dealerTotal += deck.dealCard().getCardValue();
        std::cout << "The dealer now has: " << dealerTotal << '\n';
    }

    // If dealer busted, player wins
    if (dealerTotal > 21)
        return true;

    return (playerTotal > dealerTotal);
}

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

    Deck deck;
    deck.shuffleDeck();

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

    return 0;
}
```


9.1 -- Introduction to operator overloading

Index

8.16 -- Timing your code