

## 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:

```

1  #include <iostream>
2
3
4  int main()
5  {
6      Point2d first;
7      Point2d second(3.0, 4.0);
8      first.print();
9      second.print();
10
11     return 0;
12 }

```

This should print:

```

Point2d(0, 0);
Point2d(3, 4);

```

### Hide Solution

```

1  #include <iostream>
2
3  class Point2d
4  {
5  private:
6      double m_x;
7      double m_y;
8
9  public:
10     Point2d(double x = 0.0, double y = 0.0)
11         : m_x(x), m_y(y)
12     {
13     }
14
15     void print() const
16     {
17         std::cout << "Point2d(" << m_x << ", " << m_y << ")\n";
18     }
19 };
20
21
22 int main()
23 {
24     Point2d first;
25     Point2d second(3.0, 4.0);
26     first.print();
27     second.print();
28
29     return 0;
30 }

```

1b) Now add a member function named `distanceTo` that takes another `Point2d` as a parameter, and calculates the distance between them. Given two points  $(x_1, y_1)$  and  $(x_2, y_2)$ , the distance between them can be calculated as  $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ . The `sqrt` function lives in header `cmath`.

The following program should run:

```

1  int main()
2  {
3      Point2d first;
4      Point2d second(3.0, 4.0);
5      first.print();
6      second.print();
7      std::cout << "Distance between two points: " << first.distanceTo(second) << '\n';
8
9      return 0;
10 }

```

This should print:

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

### Hide Solution

```
1  #include <cmath>
2  #include <iostream>
3
4  class Point2d
5  {
6  private:
7      double m_x;
8      double m_y;
9
10 public:
11     Point2d(double x = 0.0, double y = 0.0)
12         : m_x(x), m_y(y)
13     {
14     }
15
16     void print() const
17     {
18         std::cout << "Point2d(" << m_x << " , " << m_y << ")\n";
19     }
20
21     double distanceTo(const Point2d & other) const
22     {
23         return sqrt((m_x - other.m_x)*(m_x - other.m_x) + (m_y - other.m_y)*(m_y - other.m_y));
24     }
25 };
26
27 int main()
28 {
29     Point2d first;
30     Point2d second(3.0, 4.0);
31     first.print();
32     second.print();
33     std::cout << "Distance between two points: " << first.distanceTo(second) << '\n';
34
35     return 0;
36 }
```

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:

```
1  int main()
2  {
3      Point2d first;
4      Point2d second(3.0, 4.0);
5      first.print();
6      second.print();
7      std::cout << "Distance between two points: " << distanceFrom(first, second) << '\n';
8
9      return 0;
10 }
```

This should print:

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

## Hide Solution

```
1  #include <cmath>
2  #include <iostream>
3
4  class Point2d
5  {
6  private:
7      double m_x;
8      double m_y;
9
10 public:
11     Point2d(double x = 0.0, double y = 0.0)
12         : m_x(x), m_y(y)
13     {
14     }
15
16     void print() const
17     {
18         std::cout << "Point2d(" << m_x << " , " << m_y << ")\n";
19     }
20
21     friend double distanceFrom(const Point2d &x, const Point2d &y);
22
23 };
24
25 double distanceFrom(const Point2d &x, const Point2d &y)
26 {
27     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));
28 }
29
30 int main()
31 {
32     Point2d first;
33     Point2d second(3.0, 4.0);
34     first.print();
35     second.print();
36     std::cout << "Distance between two points: " << distanceFrom(first, second) << '\n';
37
38     return 0;
39 }
```

2) Write a destructor for this class:

```
1  class HelloWorld
2  {
3  private:
4      char *m_data;
5
6  public:
7      HelloWorld()
8      {
9          m_data = new char[14];
10         const char *init = "Hello, World!";
11         for (int i = 0; i < 14; ++i)
12             m_data[i] = init[i];
13     }
14
15     ~HelloWorld()
16     {
17         // replace this comment with your destructor implementation
18     }
19
20     void print() const
21     {
```

```

22         std::cout << m_data;
23     }
24
25 };
26
27 int main()
28 {
29     HelloWorld hello;
30     hello.print();
31
32     return 0;
33 }

```

### Hide Solution

```

1  class HelloWorld
2  {
3  private:
4      char *m_data;
5
6  public:
7      HelloWorld()
8      {
9          m_data = new char[14];
10         const char *init = "Hello, World!";
11         for (int i = 0; i < 14; ++i)
12             m_data[i] = init[i];
13     }
14
15     ~HelloWorld()
16     {
17         delete[] m_data;
18     }
19
20     void print() const
21     {
22         std::cout << m_data;
23     }
24
25 };
26
27 int main()
28 {
29     HelloWorld hello;
30     hello.print();
31
32     return 0;
33 }

```

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

```

1  enum MonsterType
2  {
3      DRAGON,
4      GOBLIN,
5      OGRE,
6      ORC,
7      SKELETON,
8      TROLL,
9      VAMPIRE,
10     ZOMBIE,

```

```
11     MAX_MONSTER_TYPES
12 };
```

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

```
1  #include <string>
2
3  enum MonsterType
4  {
5      DRAGON,
6      GOBLIN,
7      OGRE,
8      ORC,
9      SKELETON,
10     TROLL,
11     VAMPIRE,
12     ZOMBIE,
13     MAX_MONSTER_TYPES
14 };
15
16 class Monster
17 {
18 private:
19
20     MonsterType m_type;
21     std::string m_name;
22     std::string m_roar;
23     int m_hitPoints;
24 };
```

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

#### Hide Solution

```
1  #include <string>
2
3  class Monster
4  {
5  public:
6      enum MonsterType
7      {
8          DRAGON,
9          GOBLIN,
10         OGRE,
11         ORC,
12         SKELETON,
13         TROLL,
14         VAMPIRE,
15         ZOMBIE,
16         MAX_MONSTER_TYPES
17     };
18
19 private:
20
21     MonsterType m_type;
22     std::string m_name;
23     std::string m_roar;
24     int m_hitPoints;
25 };
```

3d) 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   }

```

### Hide Solution

```

1   #include <string>
2
3   class Monster
4   {
5   public:
6       enum MonsterType
7       {
8           DRAGON,
9           GOBLIN,
10          OGRE,
11          ORC,
12          SKELETON,
13          TROLL,
14          VAMPIRE,
15          ZOMBIE,
16          MAX_MONSTER_TYPES
17      };
18
19  private:
20
21      MonsterType m_type;
22      std::string m_name;
23      std::string m_roar;
24      int m_hitPoints;
25
26  public:
27      Monster(MonsterType type, std::string name, std::string roar, int hitPoints)
28          : m_type(type), m_name(name), m_roar(roar), m_hitPoints(hitPoints)
29      {
30      }
31  };
32
33
34  int main()
35  {
36      Monster skele(Monster::SKELETON, "Bones", "*rattle*", 4);
37
38      return 0;
39  }

```

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:

```

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\*

### Hide Solution

```

1  #include <iostream>
2  #include <string>
3
4  class Monster
5  {
6  public:
7      enum MonsterType
8      {
9          DRAGON,
10         GOBLIN,
11         OGRE,
12         ORC,
13         SKELETON,
14         TROLL,
15         VAMPIRE,
16         ZOMBIE,
17         MAX_MONSTER_TYPES
18     };
19
20 private:
21
22     MonsterType m_type;
23     std::string m_name;
24     std::string m_roar;
25     int m_hitPoints;
26
27 public:
28     Monster(MonsterType type, std::string name, std::string roar, int hitPoints)
29         : m_type(type), m_name(name), m_roar(roar), m_hitPoints(hitPoints)
30     {
31
32     }
33
34     std::string getTypeString() const
35     {
36         switch (m_type)
37         {
38             case DRAGON: return "dragon";
39             case GOBLIN: return "goblin";
40             case OGRE: return "ogre";
41             case ORC: return "orc";
42             case SKELETON: return "skeleton";
43             case TROLL: return "troll";
44             case VAMPIRE: return "vampire";
45             case ZOMBIE: return "zombie";
46         }
47
48         return "???";
49     }
50
51     void print() const
52     {
53         std::cout << m_name << " the " << getTypeString() << " has " << m_hitPoints << " hit points and
54 says " << m_roar << '\n';
55     }
56 };
57
58 int main()
59 {
60     Monster skele(Monster::SKELETON, "Bones", "*rattle*", 4);
61     skele.print();
62
63     return 0;
64 }

```

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 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:

```
1  int main()
2  {
3      Monster m = MonsterGenerator::generateMonster();
4      m.print();
5
6      return 0;
7  }
```

and print:

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

### Hide Solution

```
1  #include <iostream>
2  #include <string>
3
4  class Monster
5  {
6  public:
7      enum MonsterType
8      {
9          DRAGON,
10         GOBLIN,
11         OGRE,
12         ORC,
13         SKELETON,
14         TROLL,
15         VAMPIRE,
16         ZOMBIE,
17         MAX_MONSTER_TYPES
18     };
19
20     private:
21
22         MonsterType m_type;
23         std::string m_name;
24         std::string m_roar;
25         int m_hitPoints;
26
27     public:
28         Monster(MonsterType type, std::string name, std::string roar, int hitPoints)
29             : m_type(type), m_name(name), m_roar(roar), m_hitPoints(hitPoints)
30         {
31
32         }
33
34         std::string getTypeString() const
35         {
36             switch (m_type)
37             {
38                 case DRAGON: return "dragon";
39                 case GOBLIN: return "goblin";
40                 case OGRE: return "ogre";
41                 case ORC: return "orc";
42                 case SKELETON: return "skeleton";
43                 case TROLL: return "troll";
44                 case VAMPIRE: return "vampire";
45                 case ZOMBIE: return "zombie";
46             }
```

```

47         return "???";
48     }
49
50
51     void print() const
52     {
53         std::cout << m_name << " the " << getTypeString() << " has " << m_hitPoints << " hit points and
54 says " << m_roar << '\n';
55     }
56 };
57
58 class MonsterGenerator
59 {
60 public:
61     static Monster generateMonster()
62     {
63         return Monster(Monster::SKELETON, "Bones", "*rattle*", 4);
64     }
65 };
66
67 int main()
68 {
69     Monster m = MonsterGenerator::generateMonster();
70     m.print();
71
72     return 0;
73 }

```

3g) Now, MonsterGenerator needs to generate some random attributes. To do that, we'll need to make use of this handy function:

```

1 // Generate a random number between min and max (inclusive)
2 // Assumes srand() has already been called
3 int getRandomNumber(int min, int max)
4 {
5     static const double fraction = 1.0 / (static_cast<double>(RAND_MAX) + 1.0); // static used for
6 efficiency, so we only calculate this value once
7     // evenly distribute the random number across our range
8     return static_cast<int>(rand() * fraction * (max - min + 1) + min);
9 }

```

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

### Hide Solution

```

1 class MonsterGenerator
2 {
3 public:
4     // Generate a random number between min and max (inclusive)
5     // Assumes srand() has already been called
6     static int getRandomNumber(int min, int max)
7     {
8         static const double fraction = 1.0 / (static_cast<double>(RAND_MAX) + 1.0); // static used for
9 efficiency, so we only calculate this value once
10        // evenly distribute the random number across our range
11        return static_cast<int>(rand() * fraction * (max - min + 1) + min);
12    }
13
14    static Monster generateMonster()
15    {
16        return Monster(Monster::SKELETON, "Bones", "*rattle*", 4);
17    }
18 };

```

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:

```
1  #include <ctime> // for time()
2  #include <cstdlib> // for rand() and srand()
3  int main()
4  {
5      srand(static_cast<unsigned int>(time(0))); // set initial seed value to system clock
6      rand(); // If using Visual Studio, discard first random value
7
8      Monster m = MonsterGenerator::generateMonster();
9      m.print();
10
11     return 0;
12 }
```

### Hide Solution

```
1  #include <ctime> // for time()
2  #include <cstdlib> // for rand() and srand()
3  #include <iostream>
4  #include <string>
5
6  class Monster
7  {
8  public:
9      enum MonsterType
10     {
11         DRAGON,
12         GOBLIN,
13         OGRE,
14         ORC,
15         SKELETON,
16         TROLL,
17         VAMPIRE,
18         ZOMBIE,
19         MAX_MONSTER_TYPES
20     };
21
22     private:
23
24     MonsterType m_type;
25     std::string m_name;
26     std::string m_roar;
27     int m_hitPoints;
28
29     public:
30     Monster(MonsterType type, std::string name, std::string roar, int hitPoints)
31         : m_type(type), m_name(name), m_roar(roar), m_hitPoints(hitPoints)
32     {
33
34     }
35
36     std::string getTypeString() const
37     {
38         switch (m_type)
39         {
40             case DRAGON: return "dragon";
41             case GOBLIN: return "goblin";
42             case OGRE: return "ogre";
43             case ORC: return "orc";
44             case SKELETON: return "skeleton";
45             case TROLL: return "troll";
46             case VAMPIRE: return "vampire";
47             case ZOMBIE: return "zombie";
48         }
49
50         return "???";
51     }
```

```

51     }
52
53     void print() const
54     {
55         std::cout << m_name << " the " << getTypeString() << " has " << m_hitPoints << " hit points and
56 says " << m_roar << '\n';
57     }
58 };
59
60 class MonsterGenerator
61 {
62 public:
63     // Generate a random number between min and max (inclusive)
64     // Assumes srand() has already been called
65     static int getRandomNumber(int min, int max)
66     {
67         static const double fraction = 1.0 / (static_cast<double>(RAND_MAX) + 1.0); // static used for
68 efficiency, so we only calculate this value once
69         // evenly distribute the random number across our range
70         return static_cast<int>(rand() * fraction * (max - min + 1) + min);
71     }
72
73     static Monster generateMonster()
74     {
75         Monster::MonsterType type = static_cast<Monster::MonsterType>(getRandomNumber(0, Monster::MAX_M
76 ONSTER_TYPES - 1));
77         int hitPoints = getRandomNumber(1, 100);
78
79         static std::string s_names[6]{ "Blarg", "Moog", "Pksh", "Tyrn", "Mort", "Hans" };
80         static std::string s_roars[6]{ "**ROAR**", "**peep**", "**squeal**", "**whine**", "**hum**", "**burp**"};
81
82         return Monster(type, s_names[getRandomNumber(0, 5)], s_roars[getRandomNumber(0, 5)], hitPoints)
83 ;
84     }
85 };
86
87 int main()
88 {
89     srand(static_cast<unsigned int>(time(0))); // set initial seed value to system clock
90     rand(); // If using Visual Studio, discard first random value
91
92     Monster m = MonsterGenerator::generateMonster();
93     m.print();
94
95     return 0;
96 }

```

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:

```

1  #include <iostream>
2  #include <array>
3  #include <ctime> // for time()
4  #include <cstdlib> // for rand() and srand()
5
6  enum CardSuit
7  {
8      SUIT_CLUB,
9      SUIT_DIAMOND,
10     SUIT_HEART,

```

```

11     SUIT_SPADE,
12     MAX_SUITS
13 };
14
15 enum CardRank
16 {
17     RANK_2,
18     RANK_3,
19     RANK_4,
20     RANK_5,
21     RANK_6,
22     RANK_7,
23     RANK_8,
24     RANK_9,
25     RANK_10,
26     RANK_JACK,
27     RANK_QUEEN,
28     RANK_KING,
29     RANK_ACE,
30     MAX_RANKS
31 };
32
33 struct Card
34 {
35     CardRank rank;
36     CardSuit suit;
37 };
38
39 void printCard(const Card &card)
40 {
41     switch (card.rank)
42     {
43         case RANK_2:         std::cout << '2'; break;
44         case RANK_3:         std::cout << '3'; break;
45         case RANK_4:         std::cout << '4'; break;
46         case RANK_5:         std::cout << '5'; break;
47         case RANK_6:         std::cout << '6'; break;
48         case RANK_7:         std::cout << '7'; break;
49         case RANK_8:         std::cout << '8'; break;
50         case RANK_9:         std::cout << '9'; break;
51         case RANK_10:        std::cout << 'T'; break;
52         case RANK_JACK:      std::cout << 'J'; break;
53         case RANK_QUEEN:     std::cout << 'Q'; break;
54         case RANK_KING:      std::cout << 'K'; break;
55         case RANK_ACE:       std::cout << 'A'; break;
56     }
57
58     switch (card.suit)
59     {
60         case SUIT_CLUB:      std::cout << 'C'; break;
61         case SUIT_DIAMOND:   std::cout << 'D'; break;
62         case SUIT_HEART:     std::cout << 'H'; break;
63         case SUIT_SPADE:     std::cout << 'S'; break;
64     }
65 }
66
67 void printDeck(const std::array<Card, 52> deck)
68 {
69     for (const auto &card : deck)
70     {
71         printCard(card);
72         std::cout << ' ';
73     }
74
75     std::cout << '\n';
76 }
77

```

```

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;
211 }

```

Holy moly! Where do we even begin? Don't worry, 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:

```
1  #include <iostream>
2
3  int main()
4  {
5      const Card cardQueenHearts(Card::RANK_QUEEN, Card::SUIT_HEART);
6      cardQueenHearts.printCard();
7      std::cout << " has the value " << cardQueenHearts.getCardValue() << '\n';
8
9      return 0;
10 }
```

#### Hide Solution

```
1  #include <iostream>
2
3  class Card
4  {
5  public:
6      enum CardSuit
7      {
8          SUIT_CLUB,
9          SUIT_DIAMOND,
10         SUIT_HEART,
11         SUIT_SPADE,
12         MAX_SUITS
13     };
14
15     enum CardRank
16     {
17         RANK_2,
18         RANK_3,
19         RANK_4,
20         RANK_5,
21         RANK_6,
22         RANK_7,
23         RANK_8,
24         RANK_9,
25         RANK_10,
26         RANK_JACK,
27         RANK_QUEEN,
28         RANK_KING,
29         RANK_ACE,
30         MAX_RANKS
31     };
32 }
```



```

33 private:
34     CardRank m_rank;
35     CardSuit m_suit;
36
37 public:
38     Card(CardRank rank=MAX_RANKS, CardSuit suit=MAX_SUITS) :
39         m_rank(rank), m_suit(suit)
40     {
41
42     }
43
44     void printCard() const
45     {
46         switch (m_rank)
47         {
48             case RANK_2:         std::cout << '2'; break;
49             case RANK_3:         std::cout << '3'; break;
50             case RANK_4:         std::cout << '4'; break;
51             case RANK_5:         std::cout << '5'; break;
52             case RANK_6:         std::cout << '6'; break;
53             case RANK_7:         std::cout << '7'; break;
54             case RANK_8:         std::cout << '8'; break;
55             case RANK_9:         std::cout << '9'; break;
56             case RANK_10:        std::cout << 'T'; break;
57             case RANK_JACK:      std::cout << 'J'; break;
58             case RANK_QUEEN:     std::cout << 'Q'; break;
59             case RANK_KING:      std::cout << 'K'; break;
60             case RANK_ACE:       std::cout << 'A'; break;
61         }
62
63         switch (m_suit)
64         {
65             case SUIT_CLUB:      std::cout << 'C'; break;
66             case SUIT_DIAMOND:   std::cout << 'D'; break;
67             case SUIT_HEART:     std::cout << 'H'; break;
68             case SUIT_SPADE:     std::cout << 'S'; break;
69         }
70     }
71
72     int getCardValue() const
73     {
74         switch (m_rank)
75         {
76             case RANK_2:         return 2;
77             case RANK_3:         return 3;
78             case RANK_4:         return 4;
79             case RANK_5:         return 5;
80             case RANK_6:         return 6;
81             case RANK_7:         return 7;
82             case RANK_8:         return 8;
83             case RANK_9:         return 9;
84             case RANK_10:        return 10;
85             case RANK_JACK:      return 10;
86             case RANK_QUEEN:     return 10;
87             case RANK_KING:      return 10;
88             case RANK_ACE:       return 11;
89         }
90
91         return 0;
92     }
93 };
94
95 int main()
96 {
97     const Card cardQueenHearts(Card::RANK_QUEEN, Card::SUIT_HEART);
98     cardQueenHearts.printCard();
99     std::cout << " has the value " << cardQueenHearts.getCardValue() << '\n';

```

```

100
101     return 0;
102 }

```

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 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.

```

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

```

The following test program should compile:

```

1  #include <iostream>
2  #include <ctime> // for time()
3  #include <cstdlib> // for rand() and srand()
4
5  int main()
6  {
7      srand(static_cast<unsigned int>(time(0))); // set initial seed value to system clock
8      rand(); // If using Visual Studio, discard first random value
9
10     Deck deck;
11     deck.printDeck();
12     deck.shuffleDeck();
13     deck.printDeck();
14
15     return 0;
16 }

```

### Hide Solution

```

1  #include <iostream>
2  #include <array>
3  #include <ctime> // for time()
4  #include <cstdlib> // for rand() and srand()
5
6  class Card
7  {
8  public:
9      enum CardSuit
10     {
11         SUIT_CLUB,
12         SUIT_DIAMOND,
13         SUIT_HEART,
14         SUIT_SPADE,
15         MAX_SUITS
16     };
17
18     enum CardRank
19     {
20         RANK_2,
21         RANK_3,
22         RANK_4,
23         RANK_5,
24         RANK_6,
25         RANK_7,
26         RANK_8,
27         RANK_9,
28         RANK_10,
29         RANK_JACK,
30         RANK_QUEEN,

```

```

31     RANK_KING,
32     RANK_ACE,
33     MAX_RANKS
34 };
35
36 private:
37     CardRank m_rank;
38     CardSuit m_suit;
39
40 public:
41
42     Card(CardRank rank=MAX_RANKS, CardSuit suit=MAX_SUITS) :
43         m_rank(rank), m_suit(suit)
44     {
45
46     }
47
48     void printCard() const
49     {
50         switch (m_rank)
51         {
52             case RANK_2:         std::cout << '2'; break;
53             case RANK_3:         std::cout << '3'; break;
54             case RANK_4:         std::cout << '4'; break;
55             case RANK_5:         std::cout << '5'; break;
56             case RANK_6:         std::cout << '6'; break;
57             case RANK_7:         std::cout << '7'; break;
58             case RANK_8:         std::cout << '8'; break;
59             case RANK_9:         std::cout << '9'; break;
60             case RANK_10:        std::cout << 'T'; break;
61             case RANK_JACK:      std::cout << 'J'; break;
62             case RANK_QUEEN:     std::cout << 'Q'; break;
63             case RANK_KING:      std::cout << 'K'; break;
64             case RANK_ACE:       std::cout << 'A'; break;
65         }
66
67         switch (m_suit)
68         {
69             case SUIT_CLUB:      std::cout << 'C'; break;
70             case SUIT_DIAMOND:   std::cout << 'D'; break;
71             case SUIT_HEART:     std::cout << 'H'; break;
72             case SUIT_SPADE:     std::cout << 'S'; break;
73         }
74     }
75
76     int getCardValue() const
77     {
78         switch (m_rank)
79         {
80             case RANK_2:         return 2;
81             case RANK_3:         return 3;
82             case RANK_4:         return 4;
83             case RANK_5:         return 5;
84             case RANK_6:         return 6;
85             case RANK_7:         return 7;
86             case RANK_8:         return 8;
87             case RANK_9:         return 9;
88             case RANK_10:        return 10;
89             case RANK_JACK:      return 10;
90             case RANK_QUEEN:     return 10;
91             case RANK_KING:      return 10;
92             case RANK_ACE:       return 11;
93         }
94
95         return 0;
96     }
97 };

```

```

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 for
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>(suit));
129                 ++card;
130             }
131     }
132
133     void printDeck() const
134     {
135         for (const auto &card : m_deck)
136         {
137             card.printCard();
138             std::cout << ' ';
139         }
140
141         std::cout << '\n';
142     }
143
144     void shuffleDeck()
145     {
146         // Step through each card in the deck
147         for (int index = 0; index < 52; ++index)
148         {
149             // Pick a random card, any card
150             int swapIndex = getRandomNumber(0, 51);
151             // Swap it with the current card
152             swapCard(m_deck[index], m_deck[swapIndex]);
153         }
154     }
155
156 };
157
158 int main()
159 {
160     srand(static_cast<unsigned int>(time(0))); // set initial seed value to system clock
161     rand(); // If using Visual Studio, discard first random value
162
163     Deck deck;

```

```

165     deck.printDeck();
166     deck.shuffleDeck();
167     deck.printDeck();
168
169     return 0;
170 }

```

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:

```

1  int main()
2  {
3      srand(static_cast<unsigned int>(time(0))); // set initial seed value to system clock
4      rand(); // If using Visual Studio, discard first random value
5
6      Deck deck;
7      deck.shuffleDeck();
8      deck.printDeck();
9      std::cout << "The first card has value: " << deck.dealCard().getCardValue() << '\n';
10     std::cout << "The second card has value: " << deck.dealCard().getCardValue() << '\n';
11
12     return 0;
13 }

```

#### Hide Solution

```

1  #include <iostream>
2  #include <array>
3  #include <ctime> // for time()
4  #include <cstdlib> // for rand() and srand()
5  #include <cassert> // for assert()
6
7  class Card
8  {
9  public:
10     enum CardSuit
11     {
12         SUIT_CLUB,
13         SUIT_DIAMOND,
14         SUIT_HEART,
15         SUIT_SPADE,
16         MAX_SUITS
17     };
18
19     enum CardRank
20     {
21         RANK_2,
22         RANK_3,
23         RANK_4,
24         RANK_5,
25         RANK_6,
26         RANK_7,
27         RANK_8,
28         RANK_9,
29         RANK_10,
30         RANK_JACK,
31         RANK_QUEEN,
32         RANK_KING,
33         RANK_ACE,
34         MAX_RANKS
35     };
36
37     private:

```

```

38     CardRank m_rank;
39     CardSuit m_suit;
40
41 public:
42
43     Card(CardRank rank=MAX_RANKS, CardSuit suit=MAX_SUITS) :
44         m_rank(rank), m_suit(suit)
45     {
46
47     }
48
49     void printCard() const
50     {
51         switch (m_rank)
52         {
53             case RANK_2:         std::cout << '2'; break;
54             case RANK_3:         std::cout << '3'; break;
55             case RANK_4:         std::cout << '4'; break;
56             case RANK_5:         std::cout << '5'; break;
57             case RANK_6:         std::cout << '6'; break;
58             case RANK_7:         std::cout << '7'; break;
59             case RANK_8:         std::cout << '8'; break;
60             case RANK_9:         std::cout << '9'; break;
61             case RANK_10:        std::cout << 'T'; break;
62             case RANK_JACK:      std::cout << 'J'; break;
63             case RANK_QUEEN:     std::cout << 'Q'; break;
64             case RANK_KING:      std::cout << 'K'; break;
65             case RANK_ACE:       std::cout << 'A'; break;
66         }
67
68         switch (m_suit)
69         {
70             case SUIT_CLUB:      std::cout << 'C'; break;
71             case SUIT_DIAMOND:   std::cout << 'D'; break;
72             case SUIT_HEART:     std::cout << 'H'; break;
73             case SUIT_SPADE:     std::cout << 'S'; break;
74         }
75     }
76
77     int getCardValue() const
78     {
79         switch (m_rank)
80         {
81             case RANK_2:         return 2;
82             case RANK_3:         return 3;
83             case RANK_4:         return 4;
84             case RANK_5:         return 5;
85             case RANK_6:         return 6;
86             case RANK_7:         return 7;
87             case RANK_8:         return 8;
88             case RANK_9:         return 9;
89             case RANK_10:        return 10;
90             case RANK_JACK:      return 10;
91             case RANK_QUEEN:     return 10;
92             case RANK_KING:      return 10;
93             case RANK_ACE:       return 11;
94         }
95
96         return 0;
97     }
98 };
99
100 class Deck
101 {
102 private:
103     std::array<Card, 52> m_deck;
104     int m_cardIndex = 0;

```

```

105 // Generate a random number between min and max (inclusive)
106 // Assumes srand() has already been called
107 static int getRandomNumber(int min, int max)
108 {
109     static const double fraction = 1.0 / (static_cast<double>(RAND_MAX) + 1.0); // static used for
110 efficiency, so we only calculate this value once
111 // evenly distribute the random number across our range
112 return static_cast<int>(rand() * fraction * (max - min + 1) + min);
113 }
114
115 static void swapCard(Card &a, Card &b)
116 {
117     Card temp = a;
118     a = b;
119     b = temp;
120 }
121
122 public:
123 Deck()
124 {
125     int card = 0;
126     for (int suit = 0; suit < Card::MAX_SUITS; ++suit)
127         for (int rank = 0; rank < Card::MAX_RANKS; ++rank)
128         {
129             m_deck[card] = Card(static_cast<Card::CardRank>(rank), static_cast<Card::CardSuit>(suit));
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     std::cout << '\n';
142 }
143
144 void shuffleDeck()
145 {
146     // Step through each card in the deck
147     for (int index = 0; index < 52; ++index)
148     {
149         // Pick a random card, any card
150         int swapIndex = getRandomNumber(0, 51);
151         // Swap it with the current card
152         swapCard(m_deck[index], m_deck[swapIndex]);
153     }
154 }
155
156 m_cardIndex = 0; // start a new deal
157 }
158
159 const Card& dealCard()
160 {
161     assert (m_cardIndex < 52);
162     return m_deck[m_cardIndex++];
163 }
164
165 };
166
167 int main()
168 {
169     srand(static_cast<unsigned int>(time(0))); // set initial seed value to system clock
170 }
171

```

```

172     rand(); // If using Visual Studio, discard first random value
173
174     Deck deck;
175     deck.shuffleDeck();
176     deck.printDeck();
177     std::cout << "The first card has value: " << deck.dealCard().getCardValue() << '\n';
178     std::cout << "The second card has value: " << deck.dealCard().getCardValue() << '\n';
179
180     return 0;
181 }

```

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

```

1  #include <iostream>
2  #include <array>
3  #include <ctime> // for time()
4  #include <cstdlib> // for rand() and srand()
5  #include <cassert> // for assert()
6
7  class Card
8  {
9  public:
10     enum CardSuit
11     {
12         SUIT_CLUB,
13         SUIT_DIAMOND,
14         SUIT_HEART,
15         SUIT_SPADE,
16         MAX_SUITS
17     };
18
19     enum CardRank
20     {
21         RANK_2,
22         RANK_3,
23         RANK_4,
24         RANK_5,
25         RANK_6,
26         RANK_7,
27         RANK_8,
28         RANK_9,
29         RANK_10,
30         RANK_JACK,
31         RANK_QUEEN,
32         RANK_KING,
33         RANK_ACE,
34         MAX_RANKS
35     };
36
37     private:
38         CardRank m_rank;
39         CardSuit m_suit;
40
41     public:
42
43         Card(CardRank rank=MAX_RANKS, CardSuit suit=MAX_SUITS) :
44             m_rank(rank), m_suit(suit)
45         {
46
47         }
48
49         void printCard() const
50         {
51             switch (m_rank)

```



```

52     {
53         case RANK_2:      std::cout << '2'; break;
54         case RANK_3:      std::cout << '3'; break;
55         case RANK_4:      std::cout << '4'; break;
56         case RANK_5:      std::cout << '5'; break;
57         case RANK_6:      std::cout << '6'; break;
58         case RANK_7:      std::cout << '7'; break;
59         case RANK_8:      std::cout << '8'; break;
60         case RANK_9:      std::cout << '9'; break;
61         case RANK_10:     std::cout << 'T'; break;
62         case RANK_JACK:    std::cout << 'J'; break;
63         case RANK_QUEEN:   std::cout << 'Q'; break;
64         case RANK_KING:    std::cout << 'K'; break;
65         case RANK_ACE:     std::cout << 'A'; break;
66     }
67
68     switch (m_suit)
69     {
70         case SUIT_CLUB:    std::cout << 'C'; break;
71         case SUIT_DIAMOND: std::cout << 'D'; break;
72         case SUIT_HEART:   std::cout << 'H'; break;
73         case SUIT_SPADE:   std::cout << 'S'; break;
74     }
75 }
76
77 int getCardValue() const
78 {
79     switch (m_rank)
80     {
81         case RANK_2:      return 2;
82         case RANK_3:      return 3;
83         case RANK_4:      return 4;
84         case RANK_5:      return 5;
85         case RANK_6:      return 6;
86         case RANK_7:      return 7;
87         case RANK_8:      return 8;
88         case RANK_9:      return 9;
89         case RANK_10:     return 10;
90         case RANK_JACK:    return 10;
91         case RANK_QUEEN:   return 10;
92         case RANK_KING:    return 10;
93         case RANK_ACE:     return 11;
94     }
95
96     return 0;
97 }
98 };
99
100 class Deck
101 {
102 private:
103     std::array<Card, 52> m_deck;
104     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 for
111         // 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>(suit));
131                 ++card;
132             }
133     }
134
135     void printDeck() const
136     {
137         for (const auto &card : m_deck)
138         {
139             card.printCard();
140             std::cout << ' ';
141         }
142
143         std::cout << '\n';
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         m_cardIndex = 0; // start a new deal
158     }
159
160     const Card& dealCard()
161     {
162         assert (m_cardIndex < 52);
163         return m_deck[m_cardIndex++];
164     }
165 };
166
167 char getPlayerChoice()
168 {
169     std::cout << "(h) to hit, or (s) to stand: ";
170     char choice;
171     do
172     {
173         std::cin >> choice;
174     } while (choice != 'h' && choice != 's');
175
176     return choice;
177 }
178
179 bool playBlackjack(Deck &deck)
180 {
181     int playerTotal = 0;

```

```

186     int dealerTotal = 0;
187
188     // Deal the dealer one card
189     dealerTotal += deck.dealCard().getCardValue();
190     std::cout << "The dealer is showing: " << dealerTotal << '\n';
191
192     // Deal the player two cards
193     playerTotal += deck.dealCard().getCardValue();
194     playerTotal += deck.dealCard().getCardValue();
195
196     // Player goes first
197     while (1)
198     {
199         std::cout << "You have: " << playerTotal << '\n';
200         char choice = getPlayerChoice();
201         if (choice == 's')
202             break;
203
204         playerTotal += deck.dealCard().getCardValue();
205
206         // See if the player busted
207         if (playerTotal > 21)
208             return false;
209     }
210
211     // If player hasn't busted, dealer goes until he has at least 17 points
212     while (dealerTotal < 17)
213     {
214         dealerTotal += deck.dealCard().getCardValue();
215         std::cout << "The dealer now has: " << dealerTotal << '\n';
216     }
217
218     // If dealer busted, player wins
219     if (dealerTotal > 21)
220         return true;
221
222     return (playerTotal > dealerTotal);
223 }
224
225 int main()
226 {
227     srand(static_cast<unsigned int>(time(0))); // set initial seed value to system clock
228     rand(); // If using Visual Studio, discard first random value
229
230     Deck deck;
231     deck.shuffleDeck();
232
233     if (playBlackjack(deck))
234         std::cout << "You win!\n";
235     else
236         std::cout << "You lose!\n";
237
238     return 0;
239 }

```



**9.1 -- Introduction to operator overloading**



**Index**



**8.16 -- Timing your code**