# 11.x — Chapter 11 comprehensive quiz

BY ALEX ON OCTOBER 29TH, 2016 | LAST MODIFIED BY ALEX ON JULY 9TH, 2018

**Summary**

Inheritance allows us to model an is-a relationship between two objects. The object being inherited from is called the parent class, base class, or superclass. The object doing the inheriting is called the child class, derived class, or subclass.

When a derived class inherits from a base class, the derived class acquires all of the members of the base class.

When a derived class is constructed, the base portion of the class is constructed first, and then the derived portion is constructed. In more detail:

1. Memory for the derived class is set aside (enough for both the base and derived portions).
2. The appropriate derived class constructor is called.
3. The base class object is constructed first using the appropriate base class constructor. If no base class constructor is specified, the default constructor will be used.
4. The initialization list of the derived class initializes members of the derived class.
5. The body of the derived class constructor executes.
6. Control is returned to the caller.

Destruction happens in the opposite order, from most-derived to most-base class.

C++ has 3 access specifiers: public, private, and protected. The protected access specifier allows the class the member belongs to, friends, and derived classes to access the protected member, but not the public.

Classes can inherit from another class publicly, privately, or protectedly. Classes almost always inherit publically.

Here's a table of all of the access specifier and inheritance types combinations:

| Access specifier in base class | Access specifier when inherited publicly | Access specifier when inherited privately | Access specifier when inherited protectedly |
|---|---|---|---|
| Public | Public | Private | Protected |
| Private | Inaccessible | Inaccessible | Inaccessible |
| Protected | Protected | Private | Protected |

Derived classes can add new functions, change the way functions that exist in the base class work in the derived class, change an inherited member's access level, or hide functionality.

Multiple inheritance enables a derived class to inherit members from more than one parent. You should avoid multiple inheritance as much as possible.

**Quiz Time**

1) For each of the following programs, determine what they output, or if they would not compile, indicate why. This exercise is meant to be done by inspection, so do not compile these (otherwise the answers are trivial).

1a)

```
#include <iostream>

class Base
{
public:
    Base()
    {
        std::cout << "Base()\n";
    }
    ~Base()
    {
```

```
        std::cout << "~Base()\n";
    }
};

class Derived: public Base
{
public:
    Derived()
    {
        std::cout << "Derived()\n";
    }
    ~Derived()
    {
        std::cout << "~Derived()\n";
    }
};

int main()
{
    Derived d;
}
```

**Hide Solution**

Construction happens in order from most-Parent to most-Child. Destruction happens in the opposite order.

```
Base()
Derived()
~Derived()
~Base()
```

1b)

```
#include <iostream>

class Base
{
public:
    Base()
    {
        std::cout << "Base()\n";
    }
    ~Base()
    {
        std::cout << "~Base()\n";
    }
};

class Derived: public Base
{
public:
    Derived()
    {
        std::cout << "Derived()\n";
    }
    ~Derived()
    {
        std::cout << "~Derived()\n";
    }
};

int main()
{
    Derived d;
    Base b;
}
```

Hint: Local variables are destroyed in the opposite order of definition.

**Hide Solution**

First we construct d, which prints:

```
Base()
Derived()
```

Then we construct b, which prints:

```
Base()
```

Then we destruct b, which prints:

```
~Base()
```

Then we destruct d, which prints:

```
~Derived()
~Base()
```

1c)

```cpp
#include <iostream>

class Base
{
private:
    int m_x;
public:
    Base(int x): m_x(x)
    {
        std::cout << "Base()\n";
    }
    ~Base()
    {
        std::cout << "~Base()\n";
    }

    void print() { std::cout << "Base: " << m_x << '\n';  }
};

class Derived: public Base
{
public:
    Derived(int y):  Base(y)
    {
        std::cout << "Derived()\n";
    }
    ~Derived()
    {
        std::cout << "~Derived()\n";
    }

    void print() { std::cout << "Derived: " << m_x << '\n'; }
};

int main()
{
    Derived d(5);
    d.print();
}
```

**Hide Solution**

Doesn't compile, Derived::print() can't access private member m_x

1d)

```
#include <iostream>

class Base
{
protected:
    int m_x;
public:
    Base(int x): m_x(x)
    {
        std::cout << "Base()\n";
    }
    ~Base()
    {
        std::cout << "~Base()\n";
    }

    void print() { std::cout << "Base: " << m_x << '\n';  }
};

class Derived: public Base
{
public:
    Derived(int y):  Base(y)
    {
        std::cout << "Derived()\n";
    }
    ~Derived()
    {
        std::cout << "~Derived()\n";
    }

    void print() { std::cout << "Derived: " << m_x << '\n'; }
};

int main()
{
    Derived d(5);
    d.print();
}
```

**Hide Solution**

```
Base()
Derived()
Derived: 5
~Derived()
~Base()
```

1e)

```
#include <iostream>

class Base
{
protected:
    int m_x;
public:
    Base(int x): m_x(x)
    {
        std::cout << "Base()\n";
```

```
    }
    ~Base()
    {
        std::cout << "~Base()\n";
    }

    void print() { std::cout << "Base: " << m_x << '\n';  }
};

class Derived: public Base
{
public:
    Derived(int y):  Base(y)
    {
        std::cout << "Derived()\n";
    }
    ~Derived()
    {
        std::cout << "~Derived()\n";
    }

    void print() { std::cout << "Derived: " << m_x << '\n'; }
};

class D2 : public Derived
{
public:
    D2(int z): Derived(z)
    {
        std::cout << "D2()\n";
    }
    ~D2()
    {
        std::cout << "~D2()\n";
    }

        // note: no print() function here
};

int main()
{
    D2 d(5);
    d.print();
}
```

**Hide Solution**

Base()
Derived()
D2()
Derived: 5
~D2()
~Derived()
~Base()

2a) Write an Apple class and a Banana class that are derived from a common Fruit class. Fruit should have two members: a name, and a color.

The following program should run:

```
int main()
{
    Apple a("red");
    Banana b;

    std::cout << "My " << a.getName() << " is " << a.getColor() << ".\n";
```

```
    std::cout << "My " << b.getName() << " is " << b.getColor() << ".\n";

    return 0;
}
```

And produce the result:

```
My apple is red.
My banana is yellow.
```

**Hide Solution**

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

class Fruit
{
private:
    std::string m_name;
    std::string m_color;

public:
    Fruit(std::string name, std::string color)
        : m_name(name), m_color(color)
    {

    }

    std::string getName() { return m_name; }
    std::string getColor() { return m_color; }
};

class Apple: public Fruit
{
public:
    Apple(std::string color="red")
        : Fruit("apple", color)
    {
    }
};

class Banana : public Fruit
{
public:
    Banana()
        : Fruit("banana", "yellow")
    {

    }
};

int main()
{
    Apple a("red");
    Banana b;

    std::cout << "My " << a.getName() << " is " << a.getColor() << ".\n";
    std::cout << "My " << b.getName() << " is " << b.getColor() << ".\n";

    return 0;
}
```

2b) Add a new class to the previous program called GrannySmith that inherits from Apple.

The following program should run:

```
int main()
{
    Apple a("red");
    Banana b;
    GrannySmith c;

    std::cout << "My " << a.getName() << " is " << a.getColor() << ".\n";
    std::cout << "My " << b.getName() << " is " << b.getColor() << ".\n";
    std::cout << "My " << c.getName() << " is " << c.getColor() << ".\n";

    return 0;
}
```

And produce the result:

```
My apple is red.
My banana is yellow.
My granny smith apple is green.
```

**Hide Solution**

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

class Fruit
{
private:
    std::string m_name;
    std::string m_color;

public:
    Fruit(std::string name, std::string color)
        : m_name(name), m_color(color)
    {

    }

    std::string getName() { return m_name; }
    std::string getColor() { return m_color; }
};

class Apple: public Fruit
{
// The previous constructor we used for Apple had a fixed name ("apple").
// We need a new constructor for GrannySmith to use to set the name of the fruit
protected: // protected so only derived classes can access
    Apple(std::string name, std::string color)
        : Fruit(name, color)
    {
    }

public:
    Apple(std::string color="red")
        : Fruit("apple", color)
    {
    }
};

class Banana : public Fruit
{
public:
    Banana()
        : Fruit("banana", "yellow")
    {

    }
```

```
};

class GrannySmith : public Apple
{
public:
    GrannySmith()
        : Apple("granny smith apple", "green")
    {

    }
};

int main()
{
    Apple a("red");
    Banana b;
    GrannySmith c;

    std::cout << "My " << a.getName() << " is " << a.getColor() << ".\n";
    std::cout << "My " << b.getName() << " is " << b.getColor() << ".\n";
    std::cout << "My " << c.getName() << " is " << c.getColor() << ".\n";

    return 0;
}
```

3) Challenge time! The following quiz question is more difficult and lengthy. We're going to write a simple game where you fight monsters. The goal of the game is to collect as much gold as you can before you die or get to level 20.

Our program is going to consist of 3 classes: A Creature class, a Player class, and a Monster class. Player and Monster both inherit from Creature.

3a) First create the Creature class. Creatures have 5 attributes: A name (std::string), a symbol (a char), an amount of health (int), the amount of damage they do per attack (int), and the amount of gold they are carrying (int). Implement these as class members. Write a full set of getters (a get function for each member). Add three other functions: void reduceHealth(int) reduces the Creature's health by an integer amount. bool isDead() returns true when the Creature's health is 0 or less. void addGold(int) adds gold to the Creature.

The following program should run:

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

int main()
{
    Creature o("orc", 'o', 4, 2, 10);
    o.addGold(5);
    o.reduceHealth(1);
    std::cout << "The " << o.getName() << " has " << o.getHealth() << " health and is carrying " << o.g
etGold() << " gold.\n";

    return 0;
}
```

And produce the result:

```
The orc has 3 health and is carrying 15 gold.
```

**Hide Solution**

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

class Creature
{
protected:
    std::string m_name;
    char m_symbol;
```

```
    int m_health;
    int m_damage;
    int m_gold;

public:
    Creature(std::string name, char symbol, int health, int damage, int gold) :
        m_name(name), m_symbol(symbol), m_health(health), m_damage(damage), m_gold(gold)
    {
    }

    const std::string& getName() { return m_name; }
    char getSymbol() { return m_symbol; }
    int getHealth() { return m_health; }
    int getDamage() { return m_damage; }
    int getGold() { return m_gold; }

    void reduceHealth(int health) { m_health -= health; }
    bool isDead() { return m_health <= 0; }
    void addGold(int gold) { m_gold += gold; }
};

int main()
{
    Creature o("orc", 'o', 4, 2, 10);
    o.addGold(5);
    o.reduceHealth(1);
    std::cout << "The " << o.getName() << " has " << o.getHealth() << " health and is carrying " << o.g
etGold() << " gold.\n.";

    return 0;
}
```

3b) Now we're going to create the Player class. The Player class inherits from Creature. Player has one additional member, the player's level, which starts at 1. The player has a custom name (entered by the user), uses symbol '@', has 10 health, does 1 damage to start, and has no gold. Write a function called levelUp() that increases the player's level and damage by 1. Also write a getter for the level member. Finally, write a function called hasWon() that returns true if the player has reached level 20.

Write a new main() function that asks the user for their name and produces the output as follows:

```
Enter your name: Alex
Welcome, Alex.
You have 10 health and are carrying 0 gold.
```

**Hide Solution**

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

class Creature
{
protected:
    std::string m_name;
    char m_symbol;
    int m_health;
    int m_damage;
    int m_gold;

public:
    Creature(std::string name, char symbol, int health, int damage, int gold) :
        m_name(name), m_symbol(symbol), m_health(health), m_damage(damage), m_gold(gold)
    {
    }

    const std::string& getName() { return m_name; }
    char getSymbol() { return m_symbol; }
```

```
    int getHealth() { return m_health; }
    int getDamage() { return m_damage; }
    int getGold() { return m_gold; }

    void reduceHealth(int health) { m_health -= health; }
    bool isDead() { return m_health <= 0; }
    void addGold(int gold) { m_gold += gold; }
};

class Player : public Creature
{
    int m_level = 1;

public:
    Player(std::string name)
        : Creature(name, '@', 10, 1, 0)
    {
    }

    void levelUp()
    {
        ++m_level;
        ++m_damage;
    }

    int getLevel() { return m_level; }
    bool hasWon() { return m_level >= 20; }
};

int main()
{
    std::cout << "Enter your name: ";
    std::string playerName;
    std::cin >> playerName;

    Player p(playerName);
    std::cout << "Welcome, " << p.getName() << ".\n";

    std::cout << "You have " << p.getHealth() << " health and are carrying " << p.getGold() << " gol
d.\n.";

    return 0;
}
```

3c) Next up is the Monster class. Monster also inherits from Creature. Monsters have no non-inherited member variables.

First, write an empty Monster class inheriting from Creature, and then add an enum inside the Monster class named Type that contains enumerators for the 3 monsters that we'll have in this game: DRAGON, ORC, and SLIME (you'll also want a MAX_TYPES enumerator, as that will come in handy in a bit).

**Hide Solution**

```
class Monster : public Creature
{
public:
    enum Type
    {
        DRAGON,
        ORC,
        SLIME,
        MAX_TYPES
    };
};
```

3d) Each Monster type will have a different name, symbol, starting health, gold, and damage. Here is a table of stats for each monster Type:

| Type | Name | Symbol | Health | Damage | Gold |
|---|---|---|---|---|---|
| DRAGON | dragon | D | 20 | 4 | 100 |
| ORC | orc | o | 4 | 2 | 25 |
| SLIME | slime | s | 1 | 1 | 10 |

Next step is to write a Monster constructor, so we can create monsters. The Monster constructor should take a Type enum as a parameter, and then create a Monster with the appropriate stats for that kind of monster.

There are a number of different ways to implement this (some better, some worse). However in this case, because all of our monster attributes are predefined (not random), we'll use a lookup table. A lookup table is an array that holds all of the predefined attributes. We can use the lookup table to look up the attributes for a given monster as needed.

So how do we implement this lookup table? It's not hard. We just need two things. First, we need an array that contains an element for each monster Type. Each array element will contain a struct that contains all of the predefined attribute values for that Type of Monster.

Step 1: Create a struct type inside Monster named MonsterData. This struct should have a member for each attribute (name, symbol, health, damage, and gold).
Step 2: Declare an array of that struct as a static member of the class named monsterData (declare this like a normal array member, and then add the word static before it).
Step 3: Add the following code outside of the class. This is the definition for our lookup table:

```
Monster::MonsterData Monster::monsterData[Monster::MAX_TYPES]
{
    { "dragon", 'D', 20, 4, 100 },
    { "orc", 'o', 4, 2, 25 },
    { "slime", 's', 1, 1, 10 }
};
```

Now we can index this array to lookup any values we need! For example, to get a Dragon's gold, we can access monsterData[DRAGON].gold.

Use this lookup table to implement your constructor:

```
Monster(Type type): Creature(monsterData[type].name, ...)
```

The following program should compile:

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

int main()
{
    Monster m(Monster::ORC);
    std::cout << "A " << m.getName() << " (" << m.getSymbol() << ") was created.\n";
}
```

and print:

```
A orc (o) was created.
```

**Hide Solution**

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

class Creature
{
protected:
    std::string m_name;
    char m_symbol;
    int m_health;
    int m_damage;
```

```
    int m_gold;

public:
    Creature(std::string name, char symbol, int health, int damage, int gold) :
        m_name(name), m_symbol(symbol), m_health(health), m_damage(damage), m_gold(gold)
    {
    }

    const std::string& getName() { return m_name; }
    char getSymbol() { return m_symbol; }
    int getHealth() { return m_health; }
    int getDamage() { return m_damage; }
    int getGold() { return m_gold; }

    void reduceHealth(int health) { m_health -= health; }
    bool isDead() { return m_health <= 0; }
    void addGold(int gold) { m_gold += gold; }
};

class Player : public Creature
{
    int m_level = 1;

public:
    Player(std::string name)
        : Creature(name, '@', 10, 1, 0)
    {
    }

    void levelUp()
    {
        ++m_level;
        ++m_damage;
    }

    int getLevel() { return m_level; }
};

class Monster : public Creature
{
public:
    enum Type
    {
        DRAGON,
        ORC,
        SLIME,
        MAX_TYPES
    };

    struct MonsterData
    {
        const char* name;
        char symbol;
        int health;
        int damage;
        int gold;
    };

    static MonsterData monsterData[MAX_TYPES];

    Monster(Type type)
        : Creature(monsterData[type].name, monsterData[type].symbol, monsterData[type].health, monsterD
ata[type].damage, monsterData[type].gold)
    {
    }
};
```

```
Monster::MonsterData Monster::monsterData[Monster::MAX_TYPES]
{
    { "dragon", 'D', 20, 4, 100 },
    { "orc", 'o', 4, 2, 25 },
    { "slime", 's', 1, 1, 10 }
};

int main()
{
    Monster m(Monster::ORC);
    std::cout << "A " << m.getName() << " (" << m.getSymbol() << ") was created.\n";

    return 0;
}
```

3e) Finally, add a static function to Monster named getRandomMonster(). This function should pick a random number between 0 and MAX_TYPES-1 and return a monster (by value) with that Type (you'll need to static_cast the int to a Type to pass it to the Monster constructor).

You can use the following code to pick a random number:

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

// 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 eff
iciency, 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);
}
```

The following main function should run:

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

int main()
{
    srand(static_cast<unsigned int>(time(0))); // set initial seed value to system clock
    rand(); // get rid of first result

    for (int i = 0; i < 10; ++i)
    {
        Monster m = Monster::getRandomMonster();
        std::cout << "A " << m.getName() << " (" << m.getSymbol() << ") was created.\n";
    }

    return 0;
}
```

The results of this program should be randomized.

**Hide Solution**

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

// 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 ef
ficiency, 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);
}

class Creature
{
protected:
    std::string m_name;
    char m_symbol;
    int m_health;
    int m_damage;
    int m_gold;

public:
    Creature(std::string name, char symbol, int health, int damage, int gold) :
        m_name(name), m_symbol(symbol), m_health(health), m_damage(damage), m_gold(gold)
    {
    }

    char getSymbol() { return m_symbol; }
    const std::string& getName() { return m_name; }
    bool isDead() { return m_health <= 0; }
    int getGold() { return m_gold; }
    void addGold(int gold) { m_gold += gold;  }
    void reduceHealth(int health) { m_health -= health;  }
    int getHealth() { return m_health; }
    int getDamage() { return m_damage;  }
};

class Player : public Creature
{
    int m_level = 1;

public:
    Player(std::string name)
        : Creature(name, '@', 10, 1, 0)
    {
    }

    void levelUp()
    {
        ++m_level;
        ++m_damage;
    }

    int getLevel() { return m_level; }
    bool hasWon() { return m_level >= 20; }
};

class Monster : public Creature
{
public:
    enum Type
    {
        DRAGON,
        ORC,
        SLIME,
        MAX_TYPES
    };

    struct MonsterData
    {
        const char* name;
```

```
        char symbol;
        int health;
        int damage;
        int gold;
    };

    static MonsterData monsterData[MAX_TYPES];

    Monster(Type type)
        : Creature(monsterData[type].name, monsterData[type].symbol, monsterData[type].health, monster
Data[type].damage, monsterData[type].gold)
    {
    }

    static Monster getRandomMonster()
    {
        int num = getRandomNumber(0, MAX_TYPES - 1);
        return Monster(static_cast<Type>(num));
    }
};

Monster::MonsterData Monster::monsterData[Monster::MAX_TYPES]
{
    { "dragon", 'D', 20, 4, 100 },
    { "orc", 'o', 4, 2, 25 },
    { "slime", 's', 1, 1, 10 }
};

int main()
{
    srand(static_cast<unsigned int>(time(0))); // set initial seed value to system clock
    rand(); // get rid of first result

    for (int i = 0; i < 10; ++i)
    {
        Monster m = Monster::getRandomMonster();
        std::cout << "A " << m.getName() << " (" << m.getSymbol() << ") was created.\n";
    }

    return 0;
}
```

3f) We're finally set to write our game logic!

Here are the rules for the game:

- The player encounters one randomly generated monster at a time.
- For each monster, the player has two choices: (R)un or (F)ight.
- If the player decides to Run, they have a 50% chance of escaping.
- If the player escapes, they move to the next encounter will no ill effects.
- If the player does not escape, the monster gets a free attack, and the player chooses their next action.
- If the player chooses to fight, the player attacks first. The monster's health is reduced by the player's damage.
- If the monster dies, the player takes any gold the monster is carrying. The player also levels up, increasing their level and damage by 1.
- If the monster does not die, the monster attacks the player back. The player's health is reduced by the monster's damage.
- The game ends when the player has died (loss) or reached level 20 (win)
- If the player dies, the game should tell the player what level they were and how much gold they had.
- If the player wins, the game should tell the player they won, and how much gold they had

Here's a sample game session:

```
Enter your name: Alex
Welcome, Alex
You have encountered a slime (s).
(R)un or (F)ight: f
```

```
You hit the slime for 1 damage.
You killed the slime.
You are now level 2.
You found 10 gold.
You have encountered a dragon (D).
(R)un or (F)ight: r
You failed to flee.
The dragon hit you for 4 damage.
(R)un or (F)ight: r
You successfully fled.
You have encountered a orc (o).
(R)un or (F)ight: f
You hit the orc for 2 damage.
The orc hit you for 2 damage.
(R)un or (F)ight: f
You hit the orc for 2 damage.
The orc hit you for 2 damage.
(R)un or (F)ight: f
You hit the orc for 2 damage.
You are now level 3.
You found 25 gold.
You have encountered a dragon (D).
(R)un or (F)ight: r
You failed to flee.
The dragon hit you for 4 damage.
You died at level 3 and with 35 gold.
Too bad you can't take it with you!
```

Hint: Create 4 functions:

1. The main() function should handle game setup (creating the Player) and the main game loop.
2. fightMonster() handles the fight between the Player and a single Monster, including asking the player what they want to do, handling the run or fight cases.
3. attackMonster() handles the player attacking the monster, including leveling up.
4. attackPlayer() handles the monster attacking the player.

**Hide Solution**

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

// 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 ef
ficiency, 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);
}

class Creature
{
protected:
    std::string m_name;
    char m_symbol;
    int m_health;
    int m_damage;
    int m_gold;
```

```
public:
    Creature(std::string name, char symbol, int health, int damage, int gold) :
        m_name(name), m_symbol(symbol), m_health(health), m_damage(damage), m_gold(gold)
    {
    }

    char getSymbol() { return m_symbol; }
    const std::string& getName() { return m_name; }
    bool isDead() { return m_health <= 0; }
    int getGold() { return m_gold; }
    void addGold(int gold) { m_gold += gold;  }
    void reduceHealth(int health) { m_health -= health;  }
    int getHealth() { return m_health; }
    int getDamage() { return m_damage;  }
};

class Player : public Creature
{
    int m_level = 1;

public:
    Player(std::string name)
        : Creature(name, '@', 10, 1, 0)
    {
    }

    void levelUp()
    {
        ++m_level;
        ++m_damage;
    }

    int getLevel() { return m_level; }
    bool hasWon() { return m_level >= 20; }
};



class Monster : public Creature
{
public:
    enum Type
    {
        DRAGON,
        ORC,
        SLIME,
        MAX_TYPES
    };

    struct MonsterData
    {
        const char* name;
        char symbol;
        int health;
        int damage;
        int gold;
    };

    static MonsterData monsterData[MAX_TYPES];
    Type m_type;

    Monster(Type type)
        : Creature(monsterData[type].name, monsterData[type].symbol, monsterData[type].health, monster
Data[type].damage, monsterData[type].gold), m_type(type)
    {
    }
```

```
    static Monster getRandomMonster()
    {
        int num = getRandomNumber(0, MAX_TYPES - 1);
        return Monster(static_cast<Type>(num));
    }
};

Monster::MonsterData Monster::monsterData[Monster::MAX_TYPES]
{
    { "dragon", 'D', 20, 4, 100 },
    { "orc", 'o', 6, 2, 25 },
    { "slime", 's', 1, 1, 5 }
};

// This function handles the player attacking the monster
void attackMonster(Player &p, Monster &m)
{
    // If the player is dead, we can't attack the monster
    if (p.isDead())
        return;

    std::cout << "You hit the " << m.getName() << " for " << p.getDamage() << " damage.\n";

    // Reduce the monster's health by the player's damage
    m.reduceHealth(p.getDamage());

    // If the monster is now dead, level the player up
    if (m.isDead())
    {
        std::cout << "You killed the " << m.getName() << ".\n";
        p.levelUp();
        std::cout << "You are now level " << p.getLevel() << ".\n";
        std::cout << "You found " << m.getGold() << " gold.\n";
        p.addGold(m.getGold());
    }
}

// This function handles the monster attacking the player
void attackPlayer(Monster &m, Player &p)
{
    // If the monster is dead, it can't attack the player
    if (m.isDead())
        return;

    // Reduce the player's health by the monster's damage
    p.reduceHealth(m.getDamage());
    std::cout << "The " << m.getName() << " hit you for " << m.getDamage() << " damage.\n";
}

// This function handles the entire fight between a player and a randomly generated monster
void fightMonster(Player &p)
{
    // First randomly generate a monster
    Monster m = Monster::getRandomMonster();
    std::cout << "You have encountered a " << m.getName() << " (" << m.getSymbol() << ").\n";

    // While the monster isn't dead and the player isn't dead, the fight continues
    while (!m.isDead() && !p.isDead())
    {
        std::cout << "(R)un or (F)ight: ";
        char input;
        std::cin >> input;
        if (input == 'R' || input == 'r')
        {
            // 50% chance of fleeing successfully
            if (getRandomNumber(1, 2) == 1)
```

```
            {
                std::cout << "You successfully fled.\n";
                return; // success ends the encounter
            }
            else
            {
                // Failure to flee gives the monster a free attack on the player
                std::cout << "You failed to flee.\n";
                attackPlayer(m, p);
                continue;
            }
        }

        if (input == 'F' || input == 'f')
        {
            // Player attacks first, monster attacks second
            attackMonster(p, m);
            attackPlayer(m, p);
        }
    }
}

int main()
{
    srand(static_cast<unsigned int>(time(0))); // set initial seed value to system clock
    rand(); // get rid of first result

    std::cout << "Enter your name: ";
    std::string playerName;
    std::cin >> playerName;

    Player p(playerName);
    std::cout << "Welcome, " << p.getName() << '\n';

    // If the player isn't dead and hasn't won yet, the game continues
    while (!p.isDead() && !p.hasWon())
        fightMonster(p);

    // At this point, the player is either dead or has won
    if (p.isDead())
    {
        std::cout << "You died at level " << p.getLevel() << " and with " << p.getGold() << " gold.\n"
;
        std::cout << "Too bad you can't take it with you!\n";
    }
    else
    {
        std::cout << "You won the game with " << p.getGold() << " gold!\n";
    }

    return 0;
}
```


**12.1 -- Pointers and references to the base class of derived objects**

**Index**

**11.7 -- Multiple inheritance**