Chapter 11

Order of construction of derrived class

```
#include <iostream>

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

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

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

int main() {
  C c;

  return 0;
}
```

```
results:
A
B
C
~C
~B
~A
```

Calling inherited functions and overriding behavior

```
#include <iostream>

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

  void Print() { std::cout << "Base: " << 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: " << x_ << '\n'; }
};

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

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

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

```
results:

Derived: 5
```

Example & comprehensive quiz

Let's fight monsters!

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.

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/an 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/an 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/an 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 killed the orc.
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!
```

Requirements:

- 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**.
- 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.

- 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.
- Next up is the `Monster` class. **Monster also inherits from Creature**. Monsters have no non-inherited member variables. **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). **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. 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 }
};
```

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).