11.x — Chapter 11 comprehensive quiz

BY ALEX ON OCTOBER 29TH, 2016 | LAST MODIFIED BY NASCARDRIVER ON JANUARY 24TH, 2020

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 publicly.

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)

```cpp
#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;

    return 0;
}
```

Show Solution

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()
```

```cpp
{
    Derived d;
    Base b;

    return 0;
}
```

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

Show Solution

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

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

    return 0;
}
```

Show Solution

1d)

```cpp
#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() const { 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() const { std::cout << "Derived: " << m_x << '\n'; }
};

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

    return 0;
}
```

Show Solution

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() const { 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();

    return 0;
}
```

Show Solution

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.
```

Show Solution

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.
```

Show Solution

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.getGold() << " gold.\n";

    return 0;
}
```

And produce the result:

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

Show Solution

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.
```

Show Solution

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).

Show Solution

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.

Since all the properties of monsters are known at compile-time, we can declare make the lookup a `static constexpr` member of `Monster`. The definition of the lookup table is as follows:

```cpp
// As a private member of Monster.
// Note: If you're using std::string in MonsterData, you can't make
// monsterData constexpr. Remove constexpr and define the array outside of the class.
static constexpr std::array<MonsterData, Monster::MAX_TYPES> monsterData{
  { { "dragon", 'D', 20, 4, 100 },
    { "orc", 'o', 4, 2, 25 },
    { "slime", 's', 1, 1, 10 } }
};
```

A reminder

If you get an error in the declaration of `monsterData`, make sure your compiler is C++17 capable by trying to compiling the example in lesson **0.12 -- Configuring your compiler: Choosing a language standard**.

Now we can index this array to lookup any values we need! For example, to get a Dragon's gold, we can access `monsterData[Type::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";

    return 0;
}
```

and print:

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

Show Solution

3e) Finally, add a `static` function to Monster named `getRandomMonster()`. This function should pick a random number from `0` to `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).

Lesson **5.9 -- Random number generation** contains code you can use to pick a random number.

The following main function should run:

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

int main()
{
    std::srand(static_cast<unsigned int>(std::time(nullptr))); // set initial seed value to s
ystem clock
    std::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.

Show Solution

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 with 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.
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!
```

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.

Show Solution

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

Index

11.7 -- Multiple inheritance

C++ TUTORIAL | PRINT THIS POST

296 comments to 11.x — Chapter 11 comprehensive quiz

« Older Comments 1 2 3 4

Wayseeker
January 11, 2020 at 8:52 am · Reply

Nascardriver,

First, I can't thank you enough for the tutorials you have presented here. They are generally very complete and clear, especially given that I come to learning C++ with basically NO training in coding or programming except for a tiny bit of self-training in Basic Programming back in the early 90's when I was a teenager.

My goal--eventually--is to create my own roguelike game, which I know is a long way down the line, as I have a lot to learn. Quiz 3d here has given me my first real frustration (other than the usual that comes with learning coding). It may suggest a possible change that would help your tutorial.

I wrote my code, then tried to compile it, and I got errors. After researching through the tutorials and online for awhile, I decided to just look at the solution. I was immediately struck by "#include <string_view>", which I felt reasonably sure I'd never seen before.

Eventually, I was able to resolve the issue by searching for string_view in the comments and finding the comment by arcad on December 30, 2019 at 5:56 pm. Setting my project on Visual Studio to C++17 standards helped me solve the problem, but I wouldn't have known to do this without referring to this comment. Also, it appeared to be a discussion among people who are already familiar with std::string vs std::string_view, so, even though the code now compiles without error, I still don't understand why.

In pursuit of this, I then searched for discussions of std::string_view in your tutorials, and all I found was this reference from lesson 6.8b: "C-style strings are used in a lot of old or low-level code, because they have a very small memory footprint. Modern code should favor the use std::string and std::string_view, as those provide safe and easy access to the string."

My suggestion would be to either add a note about std::string_view to this quiz point, as you do with some other things here, or add some section prior in which you explain the difference between the two. For myself,