15.x — Chapter 15 summary and quiz

Chapter Review

Inside every (non-static) member function, the keyword **this** is a const pointer that holds the address of the current implicit object. We can have functions return `*this` by reference in order to enable **method chaining**, where several member functions can be called on the same object in a single expression.

Prefer to put your class definitions in a header file with the same name as the class. Trivial member functions (such as access functions, constructors with empty bodies, etc…) can be defined inside the class definition.

Prefer to define non-trivial member functions in a source file with the same name as the class.

A type that is defined inside a class type is called a **nested type** (or **member type**). Type aliases can also be nested.

Member functions defined inside a class template definition can use the template parameters of the class template itself. Member functions defined outside the class template definition must resupply a template parameter declaration, and should be defined (in the same file) just below the class template definition.

Static member variables are static duration members that are shared by all objects of the class. Static members exist even if no objects of the class have been instantiated. Prefer to access them using the class name, the scope resolution operator, and the members name.

Making static members `inline` allows them to be initialized inside the class definition.

Static member functions are member functions that can be called with no object. They do not have a `*this` pointer, and cannot access non-static data members.

Inside the body of a class, a **friend declaration** (using the `friend` keyword) can be used to tell the compiler that some other class or function is now a friend. A **friend** is a class or function (member or non-member) that has been granted full access to the private and protected members of another class. A **friend function** is a function (member or non-member) that can access the private and protected members of a class as though it were a member of that class. A **friend class** is a class that can access the private and protected members of another class.

Quiz time

Question #1

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

a) First, let's create an scoped 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.

Show Solution

b) 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`).

Show Solution

c) `enum class MonsterType` is specific to `Monster`, so make `MonsterType` a nested unscoped enum inside `Monster` and rename it to `Type`.

Show Solution

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

The following program should compile:

```
int main()
{
	Monster skeleton{ Monster::skeleton, "Bones", "*rattle*", 4 };

	return 0;
}
```

Show Solution

e) Now we want to be able to print our monster so we can validate it's correct. Write two functions: One called `getTypeString()` that returns the monster's type as a string, and one called `print()` that matches the output in the sample program below.

The following program should compile:

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

	Monster vampire{ Monster::vampire, "Nibblez", "*hiss*", 0 };
	vampire.print();

	return 0;
}
```

and print:

```
Bones the skeleton has 4 hit points and says *rattle*.
Nibblez the vampire is dead.
```

Show Solution

f) Now we can create a random monster generator. Let's consider how our `MonsterGenerator` will work. Ideally, we'll ask it to give us a `Monster`, and it will create a random one for us. Because `MonsterGenerator` doesn't have any state, this is a good candidate for a namespace.

Create a `MonsterGenerator` namespace. Create function within named `generate()`. This should return a `Monster`. For now, make it return `Monster{ Monster::skeleton, "Bones", "*rattle*", 4}`;

The following program should compile:

```
int main()
{
	Monster m{ MonsterGenerator::generate() };
	m.print();

	return 0;
}
```

and print:

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

Show Solution

g) Add two more functions to the `MonsterGenerator` namespace. `getName(int)` will take a number between 0 and 5 (inclusive) and return a name of your choice. `getRoar(int)` will also take a number between 0 and 5 (inclusive) and return a roar of your choice. Also update your `generate()` function to call `getName(0)` and `getRoar(0)`.

The following program should compile:

```
int main()
{
	Monster m{ MonsterGenerator::generate() };
	m.print();

	return 0;
}
```

and print:

```
Blarg the skeleton has 4 hit points and says *ROAR*
```

Your name and sound will vary based on what you chose.

Show Solution

h) Now we'll randomize our generated monster. Grab the "Random.h" code from 8.15 -- Global random numbers (Random.h) and save it as Random.h. Then use `Random::get()` to generate a random monster type, random name, random roar, and random hit points (between 1 and 100).

The following program should compile:

```
#include "Random.h"

int main()
{
	Monster m{ MonsterGenerator::generate() };
	m.print();

	return 0;
}
```

and print something like this:

```
Mort the zombie has 61 hit points and says *growl*
```

Show Solution