4.x — Chapter 4 comprehensive quiz

BY ALEX ON MAY 9TH, 2015 | LAST MODIFIED BY ALEX ON JANUARY 3RD, 2018

Quick review

We covered a lot of material in this chapter. Good job, you're doing great!

A block of statements (aka. a compound statement) is treated by the compiler as if it were a single statement. These are placed between curly brackets ({ and }) and used pretty much everywhere.

Local variables are variables defined within a function. They are created at the point of variable definition, and destroyed when the block they are declared in is exited. They can only be accessed inside the block in which they are declared.

Global variables are variables defined outside of a function. They are created when the program starts, and are destroyed when it ends. They can be used anywhere in the program. Non-const global variables should generally be avoided because they are evil.

The static keyword can be used to give a global variable internal linkage, so it can only be used in the file in which it is declared. It can also be used to give a local variable static duration, which means the local variable retains its value, even after it goes out of scope.

Namespaces are an area in which all names are guaranteed to be unique. Use of namespace is a great way to avoid naming collisions. Avoid use of "using statements" outside of functions.

Implicit type conversion happens when one type is converted into another type without using a cast. Explicit type conversion happens when one type is converted to another using a cast. In some cases, this is totally safe, and in others, data may be lost. Avoid C-style casts and use static_cast instead.

std::string offers an easy way to deal with text strings. Strings are always placed between double quotes.

Enumerated types let us define our own type where all of the possible values are enumerated. These are great for categorizing things. Enum classes work like enums but offer more type safety, and should be used instead of standard enums if your compiler is C++11 capable.

Typedefs allow us to create an alias for a type's name. Fixed width integers are implemented using typedefs. Typedefs are useful for giving simple names to complicated types.

And finally, structs offer us a way to group related variables into a single structure and access them using the member selection operator (.). Object-oriented programming builds heavily on top of these, so if you learn one thing from this chapter, make sure it's this one.

Quiz time!

Yay.

1) In designing a game, we decide we want to have monsters, because everyone likes fighting monsters. Declare a struct that represents your monster. The monster should have a type that can be one of the following: an ogre, a dragon, an orc, a giant spider, or a slime. If you're using C++11, use an enum class for this. If you're using an older compiler, use an enumeration for this.

Each individual monster should also have a name (use a std::string), as well as an amount of health that represents how much damage they can take before they die. Write a function named printMonster() that prints out all of the struct's members. Instantiate an ogre and a slime, initialize them using an initializer list, and pass them to printMonster().

Your program should produce the following output:

```
This Ogre is named Torg and has 145 health.
This Slime is named Blurp and has 23 health.
```

C++11 solution: **Hide Solution**

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

```

```cpp
// Define our different monster types as an enum
enum class MonsterType
{
    OGRE,
    DRAGON,
    ORC,
    GIANT_SPIDER,
    SLIME
};

// Our monster struct represents a single monster
struct Monster
{
    MonsterType type;
    std::string name;
    int health;
};

// Return the name of the monster's type as a string
// Since this could be used elsewhere, it's better to make this its own function
std::string getMonsterTypeString(Monster monster)
{
    if (monster.type == MonsterType::OGRE)
        return "Ogre";
    if (monster.type == MonsterType::DRAGON)
        return "Dragon";
    if (monster.type == MonsterType::ORC)
        return "Orc";
    if (monster.type == MonsterType::GIANT_SPIDER)
        return "Giant Spider";
    if (monster.type == MonsterType::SLIME)
        return "Slime";

    return "Unknown";
}

// Print our monster's stats
void printMonster(Monster monster)
{
    std::cout << "This " << getMonsterTypeString(monster) <<
        " is named " << monster.name <<
        " and has " << monster.health << " health.\n";
}

int main()
{
    Monster ogre = { MonsterType::OGRE, "Torg", 145 };
    Monster slime = { MonsterType::SLIME, "Blurp", 23 };

    printMonster(ogre);
    printMonster(slime);

    return 0;
}
```

non-C++11 solution: **Hide Solution**

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

// Define our different monster types as an enum
enum MonsterType
{
    MONSTER_OGRE,
    MONSTER_DRAGON,
    MONSTER_ORC,
    MONSTER_GIANT_SPIDER,
    MONSTER_SLIME
};

// Our monster struct represents a single monster
struct Monster
{
    MonsterType type;
    std::string name;
    int health;
};

// Return the name of the monster's type as a string
// Since this could be used elsewhere, it's better to make this its own function
std::string getMonsterTypeString(Monster monster)
{
    if (monster.type == MONSTER_OGRE)
        return "Ogre";
    if (monster.type == MONSTER_DRAGON)
        return "Dragon";
    if (monster.type == MONSTER_ORC)
        return "Orc";
    if (monster.type == MONSTER_GIANT_SPIDER)
        return "Giant Spider";
    if (monster.type == MONSTER_SLIME)
        return "Slime";

    return "Unknown";
}

// Print our monster's stats
void printMonster(Monster monster)
{
    std::cout << "This " << getMonsterTypeString(monster) <<
        " is named " << monster.name <<
        " and has " << monster.health << " health.\n";
}

int main()
{
    Monster ogre = { MONSTER_OGRE, "Torg", 145 };
    Monster slime = { MONSTER_SLIME, "Blurp", 23 };

    printMonster(ogre);
    printMonster(slime);

    return 0;
}
```

5.1 -- Control flow introduction

Index

4.8 -- The auto keyword