13.x — Chapter 13 summary and quiz

learncpp.com/cpp-tutorial/chapter-13-summary-and-quiz/

Congrats! You made it through another one. The knowledge you gained regarding structs will be useful when we get to C++'s most important topic -- classes!

Quick review

A **program-defined type** (also called a **user-defined type**) is a custom type that we can create for use in our own programs. The enumerated types and class types (including structs, classes and unions) allow for creation of program-defined types. Program-defined types must be defined before they can be used. The definition of a program-defined type is called a **type definition**. Type definitions are exempt from the one-definition rule.

An **enumeration** (also called an **enumerated type** or an **enum**) is a compound data type where every possible value is defined as a symbolic constant (called an **enumerator**). Enumerations are **distinct types**, meaning the compiler can differentiate them from other types (unlike type aliases).

Unscoped enumerations are named such because they put their enumerator names into the same scope as the enumeration definition itself (as opposed to creating a new scope region like a namespace does). Unscoped enumerations also provide a named scope region for their enumerators. Unscoped enumerations will implicitly convert to integral values.

Scoped enumerations work similarly to unscoped enumerations but won't implicitly convert to integers, and the enumerators are *only* placed into the scope region of the enumeration (not into the scope region where the enumeration is defined).

A **struct** (short for **structure**) is a program-defined data type that allows us to bundle multiple variables together into a single type. The variables that are part of the struct (or class) are called **data members** (or **member variables**). To access a specific member variable, we use the **member selection operator** (`operator.`) in between the struct variable name and the member name (for normal structs and references to structs), or the **member selection from pointer operator** (`operator->`) (for pointers to structs).

In general programming, an **aggregate data type** (also called an **aggregate**) is any type that can contain multiple data members. In C++, arrays and structs with only data members are **aggregates**.

Aggregates use a form of initialization called **aggregate initialization**, which allows us to directly initialize the members of aggregates. To do this, we provide an **initializer list** as an initializer, which is just a list of comma-separated values. Aggregate initialization does a

memberwise initialization, which means each member in the struct is initialized in the order of declaration.

In C++20, **Designated initializers** allow you to explicitly define which initialization values map to which members. The members must be initialized in the order in which they are declared in the struct, otherwise an error will result.

When we define a struct (or class) type, we can provide a default initialization value for each member as part of the type definition. This process is called **non-static member initialization**, and the initialization value is called a **default member initializer**.

For performance reasons, the compiler will sometimes add gaps into structures (this is called **padding**), so the size of a structure may be larger than the sum of the size of its members.

A **class template** is a template definition for instantiating class types (structs, classes, or unions). **Class template argument deduction (CTAD)** is a C++17 feature that allows the compiler to deduce the template type arguments from an initializer.

Quiz time

Yay!

Question #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.

Each individual monster should also have a name (use a `std::string` or `std::string_view`), 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.
```

Show Solution

Question #2

Specify whether objects of each of the given types should be passed by value, const address, or const reference. You can assume the function that takes these types as parameters doesn't modify them.

a) `char`

Show Solution

b) `std::string`

Show Solution

c) `unsigned long`

Show Solution

d) `bool`

Show Solution

e) An enumerated type

Show Solution

f)

```
struct Position
{
  double x{};
  double y{};
  double z{};
};
```

Show Solution

g)

```
struct Player
{
  int health{};
  // The Player struct is still under development.  More members will be added.
};
```

Show Solution

h) `int*`

Show Solution

i) `std::string_view`

Show Solution

Question #3

Create a class template named `Triad` that has 3 members of the same template type. Also create a function template named `print` that can print a Triad. The following program should compile:

```
int main()
{
	Triad t1{ 1, 2, 3 }; // note: uses CTAD to deduce template arguments
	print(t1);

	Triad t2{ 1.2, 3.4, 5.6 }; // note: uses CTAD to deduce template arguments
	print(t2);

	return 0;
}
```

and produce the following result:

```
[1, 2, 3][1.2, 3.4, 5.6]
```

If you are using C++17, you will need to provide a deduction guide for CTAD to work (see 13.14 -- Class template argument deduction (CTAD) and deduction guides for information on that).

Show Solution