14.x — Chapter 14 summary and quiz

ALEX DECEMBER 26, 2023

In this chapter, we explored the meat of C++ -- classes! This is the most important chapter in the tutorial series, as it sets the stage for much of what's left to come.

Chapter Review

In **procedural programming**, the focus is on creating "procedures" (which in C++ are called functions) that implement our program logic. We pass data objects to these functions, those functions perform operations on the data, and then potentially return a result to be used by the caller.

With **Object-oriented programming** (often abbreviated as OOP), the focus is on creating program-defined data types that contain both properties and a set of well-defined behaviors.

A **class invariant** is a condition that must be true throughout the lifetime of an object in order for the object to remain in a valid state. An object that has a violated class invariant is said to be in an **invalid state**, and unexpected or undefined behavior may result from further use of that object.

A **class** is a program-defined compound type that bundles both data and functions that work on that data.

Functions that belong to a class type are called **member functions**. The object that a member function is called on is often called the **implicit object**. Functions that are not member functions are called **non-member functions** to distinguish them from member functions. If your class type has no data members, prefer using a namespace instead.

A **const member function** is a member function that guarantees it will not modify the object or call any non-const member functions (as they may modify the object). A member function that does not (and will not ever) modify the state of the object should be made const, so that it can be called on both non-const and const objects.

Each member of a class type has a property called an **access level** that determines who can access that member. The access level system is sometimes informally called **access controls**. Access levels are defined on a per-class basis, not on a per-object basis.

Public members are members of a class type that do not have any restrictions on how they can be accessed. Public members can be accessed by anyone (as long as they are in scope). This includes other members of the same class. Public members can also be accessed by **the public**, which is what we call code that exists outside the members of a given class type. Examples of the public include non-member functions, as well as the members of other class types.

By default, all members of a struct are public members.

Private members are members of a class type that can only be accessed by other members of the same class.

By default, the members of a class are private. A class with private members is no longer an aggregate, and therefore can no longer use aggregate initialization. Consider naming your private members starting with an "m_" prefix to help distinguish them from the names of local variables, function parameters, and member functions.

We can explicitly set the access level of our members by using an **access specifier**. Structs should generally avoid using access specifiers so all members default to public.

An **access function** is a trivial public member function whose job is to retrieve or change the value of a private member variable. Access functions come in two flavors: getters and setters. **Getters** (also sometimes called **accessors**) are public member functions that return the value of a private member variable. **Setters** (also sometimes called **mutators**) are public member functions that set the value of a private member variable.

The **interface** of a class type defines how a user of the class type will interact with objects of the class type. Because only public members can be accessed from outside of the class type, the public members of a class type form its interface. For this reason, an interface composed of public members is sometimes called a **public interface**.

The **implementation** of a class type consists of the code that actually makes the class behave as intended. This includes both the member variables that store data, and the bodies of the member functions that contain the program logic and manipulate the member variables.

In programming, **data hiding** (also called **information hiding** or **data abstraction**) is a technique used to enforce the separation of interface and implementation by hiding the implementation of a program-defined data type from users.

The term **encapsulation** is also sometimes used to refer to data hiding. However, this term is also used to refer to the bundling of data and functions together (without regard for access controls), so its use can be ambiguous.

When defining a class, prefer to declare your public members first and your private members last. This spotlights the public interface and de-emphasizes implementation details.

A **constructor** is a special member function that is used to initialize class type objects. A matching constructor must be found in order to create a non-aggregate class type object.

A **Member initializer list** allows you to initialize your member variables from within a constructor. Member variables in a member initializer list should be listed in order that they are defined in the class. Prefer using the member initializer list to initialize your members over assigning values in the body of the constructor.

A constructor that takes no parameters (or has all default parameters) is called a **default constructor**. The default constructor is used if no initialization values are provided by the user. If a non-aggregate class type object has no user-declared constructors, the compiler will generate a default constructor (so that the class can be value or default initialized). This constructor is called an **implicit default constructor**.

Constructors are allowed to delegate initialization to another constructor from the same class type. This process is sometimes called **constructor chaining** and such constructors are called **delegating constructors**. Constructors can delegate or initialize, but not both.

A **temporary object** (sometimes called an **anonymous object** or an **unnamed object**) is an object that has no name and exists only for the duration of a single expression.

A **copy constructor** is a constructor that is used to initialize an object with an existing object of the same type. If you do not provide a copy constructor for your classes, C++ will create a public **implicit copy constructor** for you that does memberwise initialization.

The **as-if rule** says that the compiler can modify a program however it likes in order to produce more optimized code, so long as those modifications do not affect a program's "observable behavior". One exception to the as-if rule is copy elision. **Copy elision** is a compiler optimization technique that allows the compiler to remove unnecessary copying of objects. When the compiler optimizes away a call to the copy constructor, we say the constructor has been **elided**.

A function that we've written to convert a value to or from a program-defined type is called a **user-defined conversion**. A constructor that can be used to perform an implicit conversion is called a **converting constructor**. By default, all constructors are converting constructors.

We can use the **explicit** keyword to tell the compiler that a constructor should not be used as a converting constructor. Such a constructor can not be used to do copy initialization or copy list initialization, nor can it be used to do implicit conversions.

Make any constructor that accepts a single argument explicit by default. If an implicit conversion between types is both semantically equivalent and performant (such as a conversion from `std::string` to `std::string_view`), you can consider making the constructor non-explicit. Do not make copy or move constructors explicit, as these do not perform conversions.

Quiz time

Author's note

The blackjack quiz that used to be part of this lesson has been moved to lesson [17.x -- Chapter 17 summary and quiz (https://www.learncpp.com/cpp-tutorial/chapter-17-summary-and-quiz/)](https://www.learncpp.com/cpp-tutorial/chapter-17-summary-and-quiz/).

Question #1

a) Write a class named `Point2d`. `Point2d` should contain two member variables of type `double`: `m_x`, and `m_y`, both defaulted to `0.0`.

Provide a constructor and a `print()` function.

The following program should run:

```
#include <iostream>

int main()
{
    Point2d first{};
    Point2d second{ 3.0, 4.0 };

    // Point2d third{ 4.0 }; // should error if uncommented

    first.print();
    second.print();

    return 0;
}
```

This should print:

```
Point2d(0, 0)
Point2d(3, 4)
```

Show Solution (javascript:void(0))

b) Now add a member function named `distanceTo()` that takes another `Point2d` as a parameter, and calculates the distance between them. Given two points (x1, y1) and (x2, y2), the distance between them can be calculated using the formula `std::sqrt((x1 - x2)*(x1 - x2) + (y1 - y2)*(y1 - y2))`. The `std::sqrt` function lives in header `cmath`.

The following program should run:

```
#include <cmath>
#include <iostream>

int main()
{
    Point2d first{};
    Point2d second{ 3.0, 4.0 };

    first.print();
    second.print();

    std::cout << "Distance between two points: " << first.distanceTo(second) << '\n';

    return 0;
}
```

This should print:

```
Point2d(0, 0)
Point2d(3, 4)
Distance between two points: 5
```

Show Solution (javascript:void(0))

Question #2

In lesson 13.8 -- Passing and returning structs (https://www.learncpp.com/cpp-tutorial/passing-and-returning-structs/), we wrote a short program using a `Fraction` struct. The reference solution looks like this:

```
#include <iostream>

struct Fraction
{
    int numerator{ 0 };
    int denominator{ 1 };
};

Fraction getFraction()
{
    Fraction temp{};
    std::cout << "Enter a value for numerator: ";
    std::cin >> temp.numerator;
    std::cout << "Enter a value for denominator: ";
    std::cin >> temp.denominator;
    std::cout << '\n';

    return temp;
}

Fraction multiply(const Fraction& f1, const Fraction& f2)
{
    return { f1.numerator * f2.numerator, f1.denominator * f2.denominator };
}

void printFraction(const Fraction& f)
{
    std::cout << f.numerator << '/' << f.denominator << '\n';
}

int main()
{
    Fraction f1{ getFraction() };
    Fraction f2{ getFraction() };

    std::cout << "Your fractions multiplied together: ";

    printFraction(multiply(f1, f2));

    return 0;
}
```

Convert `Fraction` from a struct to a class following the standard best practices. Convert all of the functions to member functions.

Show Solution (javascript:void(0))

Question #3

In the prior quiz solution, why was the Fraction constructor made `explicit`?

Show Solution (javascript:void(0))

Question #4

Extra credit: In the Fraction quiz question, which of the member functions are probably better left as non-member functions, and why?

Show Solution (javascript:void(0))

Next lesson
15.1 The hidden "this" pointer and member function chaining

Back to table of contents

Previous lesson
14.16 Converting constructors and the explicit keyword

Leave a comment...

Name*

Email*

Notify me about replies: POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/ are connected to your provided email address.

759 COMMENTS Newest