14.2 — Introduction to classes

ALEX[1] JUNE 26, 2024

In the previous chapter, we covered structs (13.7 -- Introduction to structs, members, and member selection (https://www.learncpp.com/cpp-tutorial/introduction-to-structs-members-and-member-selection/)[2]), and discussed how they are great for bundling multiple member variables into a single object that can be initialized and passed around as a unit. In other words, structs provide a convenient package for storing and moving related data values.

Consider the following struct:

```
#include <iostream>

struct Date
{
    int day{};
    int month{};
    int year{};
};

void printDate(const Date& date)
{
    std::cout << date.day << '/' << date.month << '/' << date.year; // assume DMY
format
}

int main()
{
    Date date{ 4, 10, 21 }; // initialize using aggregate initialization
    printDate(date);        // can pass entire struct to function

    return 0;
}
```

In the above example, we create a `Date` object and then pass it to a function that prints the date. This program prints:

```
4/10/21
```

A reminder

In these tutorials, all of our structs are aggregates. We discuss aggregates in lesson 13.8 -- Struct aggregate initialization (https://www.learncpp.com/cpp-tutorial/struct-aggregate-initialization/)[3].

As useful as structs are, structs have a number of deficiencies that can present challenges when trying to build large, complex programs (especially those worked on by multiple developers).

The class invariant problem

Perhaps the biggest difficulty with structs is that they do not provide an effective way to document and enforce class invariants. In lesson 9.6 -- Assert and static_assert (https://www.learncpp.com/cpp-tutorial/assert-and-static_assert/)[4], we defined an invariant as, "a condition that must be true while some component is executing".

In the context of class types (which include structs, classes, and unions), 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.

Key insight

Using an object whose class invariant has been violated may result in unexpected or undefined behavior.

First, consider the following struct:

```
struct Pair
{
    int first {};
    int second {};
};
```

The `first` and `second` members can be independently set to any value, so `Pair` struct has no invariant.

Now consider the following almost-identical struct:

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

We know from mathematics that a fraction with a denominator of `0` is mathematically undefined (because the value of a fraction is its numerator divided by its denominator -- and division by `0` is mathematically undefined). Therefore, we want to ensure the `denominator` member of a Fraction object is never set to `0`. If it is, then that Fraction object is in an invalid state, and undefined behavior may result from further use of that object.

For example:

```cpp
#include <iostream>

struct Fraction
{
    int numerator { 0 };
    int denominator { 1 }; // class invariant: should never be 0
};

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

int main()
{
    Fraction f { 5, 0 };   // create a Fraction with a zero denominator
    printFractionValue(f); // cause divide by zero error

    return 0;
}
```

In the above example, we use a comment to document the invariant of Fraction. We also provide a default member initializer to ensure that denominator is set to `1` if the user does not provide an initialization value. This ensures our Fraction object will be valid if the user decides to value initialize a Fraction object. That's an okay start.

But nothing prevents us from explicitly violating this class invariant: When we create `Fraction f`, we use aggregate initialization to explicitly initialize the denominator to `0`. While this does not cause an immediate issue, our object is now in an invalid state, and further use of the object may cause unexpected or undefined behavior.

And that is exactly what we see later, when we call `printFractionValue(f)`: the program terminates due to a divide-by-zero error.

As an aside...

A small improvement would be to `assert(f.denominator != 0);` at the top of the body of `printFractionValue`. This adds documentation value to the code, and makes it more obvious what precondition is being violated. However, behaviorally, this doesn't really change anything. We really want to catch these problems at the source of the problem (when the member is initialized or assigned a bad value), not somewhere downstream (when the bad value is used).

Given the relative simplicitly of the Fraction example, it shouldn't be too difficult to simply avoid creating invalid Fraction objects. However, in a more complex code base that uses many structs, structs with many members, or structs whose members have complex relationships, understanding what combination of values might violate some class invariant may not be so obvious.

A more complex class invariant

The class invariant for Fraction is a simple one -- the `denominator` member cannot be `0`. That's conceptually easy to understand and not overly difficult to avoid.

Class invariants become more of a challenge when the members of a struct must have correlated values.

```cpp
#include <string>

struct Employee
{
    std::string name { };
    char firstInitial { }; // should always hold first character of `name` (or `0`)
};
```

In the above (poorly designed) struct, the character value stored in member `firstInitial` should always match the first character of `name`.

When an `Employee` object is initialized, the user is responsible for making sure the class invariant is maintained. And if `name` is ever assigned a new value, the user is also responsible for ensuring `firstInitial` is updated as well. This correlation may not be obvious to a developer using an Employee object, and even if it is, they may forget to do it.

Even if we write functions to help us create and update Employee objects (ensuring that `firstInitial` is always set from the first character of `name`), we're still relying on the user to be aware of and use these functions.

In short, relying on the user of an object to maintain class invariants is likely to result in problematic code.

Key insight

Relying on the user of an object to maintain class invariants is likely to result in problems.

Ideally, we'd love to bulletproof our class types so that an object either can't be put into an invalid state, or can signal immediately if it is (rather than letting undefined behavior occur at some random point in the future).

Structs (as aggregates) just don't have the mechanics required to solve this problem in an elegant way.

Introduction to classes

When developing C++, Bjarne Stroustrup wanted to introduce capabilities that would allow developers to create program-defined types that could be used more intuitively. He was also interested in finding elegant solutions to some of the frequent pitfalls and maintenance challenges that plague large, complex programs (such as the previously mentioned class invariant issue).

Drawing upon his experience with other programming languages (particularly Simula, the first object-oriented programming language), Bjarne was convinced that it was possible to develop a program-defined type that was general and powerful enough to be used for almost anything. In a nod to Simula, he called this type a *class*.

Just like structs, a **class** is a program-defined compound type that can have many member variables with different types.

Key insight

From a technical standpoint, structs and classes are almost identical -- therefore, any example that is implemented using a struct could be implemented using a class, or vice-versa. However, from a practical standpoint, we use structs and classes differently.

We cover both the technical and practical differences between structs and classes in lesson 14.5 -- Public and private members and access specifiers (https://www.learncpp.com/cpp-tutorial/public-and-private-members-and-access-specifiers/)[5]

Related content

We cover how classes solve the invariant problem in lesson 14.8 -- The benefits of data hiding (encapsulation) (https://www.learncpp.com/cpp-tutorial/the-benefits-of-data-hiding-encapsulation/)[6].

Defining a class

Because a class is a program-defined data type, it must be defined before it can be used. Classes are defined similarly to structs, except we use the `class` keyword instead of `struct`. For example, here is a definition for a simple employee class:

```
class Employee
{
    int m_id {};
    int m_age {};
    double m_wage {};
};
```

Related content

We discuss why member variables of a class are often prefixed with an "m_" in upcoming lesson 14.5 -- Public and private members and access specifiers (https://www.learncpp.com/cpp-tutorial/public-and-private-members-and-access-specifiers/)[5]

To demonstrate how similar classes and structs can be, the following program is equivalent to the one we presented at the top of the lesson, but `Date` is now a class instead of a struct:

```
#include <iostream>

class Date       // we changed struct to class
{
public:          // and added this line, which is called an access specifier
    int m_day{}; // and added "m_" prefixes to each of the member names
    int m_month{};
    int m_year{};
};

void printDate(const Date& date)
{
    std::cout << date.m_day << '/' << date.m_month << '/' << date.m_year;
}

int main()
{
    Date date{ 4, 10, 21 };
    printDate(date);

    return 0;
}
```

This prints:

4/10/21

Related content

We cover what an access specifier is in upcoming lesson 14.5 -- Public and private members and access specifiers (https://www.learncpp.com/cpp-tutorial/public-and-private-members-and-access-specifiers/)[5].

Most of the C++ standard library is classes

You have already been using class objects, perhaps without knowing it. Both `std::string` and `std::string_view` are defined as classes. In fact, most of the non-aliased types in the standard library are defined as classes!

Classes are really the heart and soul of C++ -- they are so foundational that C++ was originally named "C with classes"! Once you are familiar with classes, much of your time in C++ will be spent writing, testing, and using them.

Quiz time

Question #1

Given some set of values (ages, address numbers, etc...), we might want to know what the minimum and maximum values are in that set. Since the minimum and maximum values are related, we can organize them in a struct, like so:

```
1 struct minMax
2 {
3     int min; // holds the minimum value seen so far
4     int max; // holds the maximum value seen so far
5 };
```

However, as written, this struct has an unspecified class invariant. What is the invariant?

Show Solution (javascript:void(0))[7]

Next lesson
14.3 Member functions

8

Back to table of contents

9

Previous lesson
14.1 Introduction to object-oriented programming

10

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[13] are connected to your provided email address.

Notify me about replies:

POST COMMENT

30 COMMENTS

Newest

Nidhi Gupta

March 15, 2025 8:08 pm PDT

In this lesson, we also look forward into the future and incrementally develop the concepts introduced in the last section on struct by mentioning the class invariants strengths and weaknesses-from there to making struct very efficient in representing a collection of related variables to instances which prevent invalid states, such as initializing a Fraction struct with zero as a denominator. This session gives a class-invariant definition and will show how relying on users to maintain them often leads to undefined behavior. This also introduced how with classes -similar to structs but adding mechanisms to enforce that objects could be valid states, by encapsulating and applying access control. Closing the gap between structures and classes is envisioned as paving the way to more robust and maintainable programming, especially on a large scale. This lesson also tells us now that structs and classes find themselves used differently in practice, though they are structurally similar. Classes energize the building blocks of object-oriented programming in C++. Thus more detailing about access specifiers, encapsulation, and other class features will follow in the next lessons.

1 Reply

Choco

Reply to Nidhi Gupta[14] April 26, 2025 10:04 am PDT

stfu

18 Reply

PooperShooter

March 2, 2025 10:45 pm PST

```cpp
#include <iostream>

class Pokemon
{
public:
    std::string_view m_name{};
    std::string_view m_type{};
    int m_level{};
};

void pokemonInfo(const Pokemon& p)
{
    std::cout << "Pokemon: " << p.m_name
              << "\nType: " << p.m_type
              << "\nLevel: " << p.m_level << "\n\n";
}

int main()
{
    // create objects
    Pokemon breloom{ "Breloom", "Grass/Fighting", 50};

    // print the output
    pokemonInfo(breloom);

    return 0;
}
```

4 Reply

1neWrld

February 26, 2025 6:22 am PST

I'm a bit confused. Cause I'm aware that structs and classes are identical, though in the article you refer to them as program-defined data types, but isn't a class a user-defined data type? Or is it different wording that means the exact
same thing?

0 Reply

Nidhi Gupta

February 1, 2025 5:50 pm PST

One of the major features of structs in C++ is that they allow a number of variables to be packed together into one for convenience in both managing and passing around a number of pieces of data. However, they do come with some notable limitations - most notably in maintaining class invariants. Without mechanisms in the language itself for enforcing these invariants, the programmer depends on careful manual management, with a heightened risk of invalid states leading to undefined behavior. A common example of this is a Fraction struct containing a denominator field that is easy to misuse by setting it to zero, which will cause division errors. Though mitigating these risks using assertions and helper functions can be done, it does not ensure correctness during the initialization or assignment. That

is where classes come in, which are a stronger alternative that provides encapsulation, validation, and controlled access to data members through member functions. Unlike structs, which are mainly used for passive data storage, classes introduce such concepts as data hiding and encapsulation, ensuring that data is valid throughout an object's lifetime. The difference between structs and classes, although minimal at the technical level, does carry a significant number of practical implications: structs are best suited for plain data structures, while classes are ideal for defining entities with behavior, constraints, and well-defined responsibilities. Understanding this difference is crucial for designing robust, maintainable, and error-resistant C++ programs.

0 Reply

eezcurious

Reply to Nidhi Gupta [15] May 19, 2025 12:12 am PDT

pls stfu

1 Reply

Choco

Reply to Nidhi Gupta [15] April 26, 2025 10:06 am PDT

fuck u talking about

3 Reply

Aniket

August 24, 2024 9:22 am PDT

In the example provided, the values for m_day, m_month and m_year are never set, so how are we able to access these values here?

```
1  void printDate(const Date& date)
2  {
3      std::cout << date.m_day << '/' << date.m_month << '/' << date.m_year;
4  }
```

Also, can you please explain why are we using const here?

Last edited 10 months ago by Aniket

1 Reply

apt

Reply to Aniket [16] May 26, 2025 5:43 am PDT

When you declare a variable as const, you can't change its value inside functions.

0 Reply

Alex Author

Reply to Aniket [16] August 25, 2024 9:47 pm PDT

As Sid says, the members are initialized on line 18 using aggregate initialization.

We use const here because we're passing by reference. Using a const reference allows us to pass in an argument that is a modifiable lvalue, non-modifiable lvalue, or an rvalue. It also makes clear the function doesn't modify the parameter.

If we didn't use const, we could only pass in a modifiable lvalue.

This is covered in lesson https://www.learncpp.com/cpp-tutorial/pass-by-const-lvalue-reference/

2 Reply

Sid

Reply to Aniket [16] August 24, 2024 6:58 pm PDT

The execution is done in int main() and if you look at line 18, you would see we assigned values to all members.

2 Reply

Samarjeet Mohite

June 17, 2024 2:19 pm PDT

The invariant problem was mentioned above that was caused while using structs. But how does the inception of class solved this problem?

2 Reply

Alex Author

Reply to Samarjeet Mohite [17] June 19, 2024 9:45 pm PDT

This is answered later in this chapter once we've covered a few more topics foundational to classes. Keep reading.

0 Reply

Aaheed

Reply to Alex [18] June 23, 2024 2:34 am PDT

I think there should be some info on how classes solve the invariant problem as it is shown to be one of the biggest problems, and then you don't show how do classes solve it.
Atleast a link to the lesson that shows how classes solve the invariant problem should be given in this lesson.

2 Reply

Alex Author

Reply to Aaheed [19] June 26, 2024 11:01 am PDT

Done.

5 Reply

Swaminathan R
May 30, 2024 11:30 pm PDT

```
I.say("Thank You!!", &Alex);
```

6 Reply

Mohit
May 15, 2024 11:06 pm PDT

I started learning cpp 2 months ago. I studied first few chapters to get the fundamentals. However, it seemed very lengthy to go chapter by chapter. I didn't need in-depth knowledge of everything.
I would look at the remaining chapters before I could read about classes. I thought I needed to study everything in between. But today, I just jumped over to this chapter and read it. It was easy peezy and now I wonder if I need to absolutely study all these chapters in the provided order to have some fun with cpp?

I'm learning to play around and make some games with cpp and raylib.

0 Reply

Spesader
Reply to Mohit [20] May 20, 2024 1:54 pm PDT

Reading all the chapters without skipping will equip you with a very solid foundation. Like what Alex said, the chapters do indeed build on each other; you'll definitely use what you learnt in chapter 7 to understand chapter 12 for example.

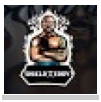
I started from chapter 0 at exactly the 2th of May, at the time of writing this I'm now at chapter 14. Unless you know all what's been covered throughout the chapters, you can definitely go back to your missed chapters and finish them all, if you're truly passionate about this stuff, it'll seem like a fun cakewalk to you. You got it.

1 Reply

Alex Author
Reply to Mohit [20] May 16, 2024 11:25 am PDT

The chapters do build on each other, so if you skip previous chapters, there may be stuff in this chapter (probably later) that doesn't make total sense. But you can always jump backwards and pick up on whatever prerequisite topics you missed at that point...

2 Reply



Rajeev dey
February 14, 2024 7:44 am PST

So, how does the problem of invariance got solved by using classes?

1 Reply

Alex Author

Reply to Rajeev dey [21] February 16, 2024 10:00 am PST

That is discussed in the next few lessons.

3 Reply

freax

January 2, 2024 12:38 pm PST

great tutorial ever

1 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/introduction-to-structs-members-and-member-selection/
3. https://www.learncpp.com/cpp-tutorial/struct-aggregate-initialization/
4. https://www.learncpp.com/cpp-tutorial/assert-and-static_assert/
5. https://www.learncpp.com/cpp-tutorial/public-and-private-members-and-access-specifiers/
6. https://www.learncpp.com/cpp-tutorial/the-benefits-of-data-hiding-encapsulation/
7. javascript:void(0)
8. https://www.learncpp.com/cpp-tutorial/member-functions/
9. https://www.learncpp.com/
10. https://www.learncpp.com/cpp-tutorial/introduction-to-object-oriented-programming/
11. https://www.learncpp.com/introduction-to-classes/
12. https://www.learncpp.com/cpp-tutorial/introduction-to-destructors/
13. https://gravatar.com/
14. https://www.learncpp.com/cpp-tutorial/introduction-to-classes/#comment-608582
15. https://www.learncpp.com/cpp-tutorial/introduction-to-classes/#comment-607264
16. https://www.learncpp.com/cpp-tutorial/introduction-to-classes/#comment-601214
17. https://www.learncpp.com/cpp-tutorial/introduction-to-classes/#comment-598538
18. https://www.learncpp.com/cpp-tutorial/introduction-to-classes/#comment-598624
19. https://www.learncpp.com/cpp-tutorial/introduction-to-classes/#comment-598720
20. https://www.learncpp.com/cpp-tutorial/introduction-to-classes/#comment-597149
21. https://www.learncpp.com/cpp-tutorial/introduction-to-classes/#comment-593630
22. https://g.ezoic.net/privacy/learncpp.com