14.9 — Introduction to constructors

ALEX[1] DECEMBER 29, 2024

When a class type is an aggregate, we can use aggregate initialization to initialize the class type directly:

```cpp
struct Foo // Foo is an aggregate
{
    int x {};
    int y {};
};

int main()
{
    Foo foo { 6, 7 }; // uses aggregate initialization

    return 0;
}
```

Aggregate inititalization does memberwise initialization (members are initialized in the order in which they are defined). So when `foo` is instantiated in the above example, `foo.x` is initialized to `6`, and `foo.y` is initialized to `7`.

Related content

We discuss the definition of an aggregate and aggregate initialization in lesson 13.8 -- Struct aggregate initialization (https://www.learncpp.com/cpp-tutorial/struct-aggregate-initialization/)[2].

However, as soon as we make any member variables private (to hide our data), our class type is no longer an aggregate (because aggregates cannot have private members). And that means we're no longer able to use aggregate initialization:

```cpp
class Foo // Foo is not an aggregate (has private members)
{
    int m_x {};
    int m_y {};
};

int main()
{
    Foo foo { 6, 7 }; // compile error: can not use aggregate initialization

    return 0;
}
```

Not allowing class types with private members to be initialized via aggregate initialization makes sense for a number of reasons:

- Aggregate initialization requires knowing about the implementation of the class (since you have to know what the members are, and what order they were defined in), which we're intentionally trying to avoid when we hide our data members.

- If our class had some kind of invariant, we'd be relying on the user to initialize the class in a way that preserves the invariant.

So then how do we initialize a class with private member variables? The error message given by the compiler for the prior example provides a clue: "error: no matching constructor for initialization of 'Foo'"

We must need a matching constructor. But what the heck is that?

Constructors

A **constructor** is a special member function that is automatically called after a non-aggregate class type object is created.

When a non-aggregate class type object is defined, the compiler looks to see if it can find an accessible constructor that is a match for the initialization values provided by the caller (if any).

- If an accessible matching constructor is found, memory for the object is allocated, and then the constructor function is called.
- If no accessible matching constructor can be found, a compilation error will be generated.

Key insight

Many new programmers are confused about whether constructors create the objects or not. They do not -- the compiler sets up the memory allocation for the object prior to the constructor call. The constructor is then called on the uninitialized object.

However, if a matching constructor cannot be found for a set of initializers, the compiler will error. So while constructors don't create objects, the lack of a matching constructor will prevent creation of an object.

Beyond determining how an object may be created, constructors generally perform two functions:

- They typically perform initialization of any member variables (via a member initialization list)
- They may perform other setup functions (via statements in the body of the constructor). This might include things such as error checking the initialization values, opening a file or database, etc...

After the constructor finishes executing, we say that the object has been "constructed", and the object should now be in a consistent, usable state.

Note that aggregates are not allowed to have constructors -- so if you add a constructor to an aggregate, it is no longer an aggregate.

Naming constructors

Unlike normal member functions, constructors have specific rules for how they must be named:

- Constructors must have the same name as the class (with the same capitalization). For template classes, this name excludes the template parameters.
- Constructors have no return type (not even `void`).

Because constructors are typically part of the interface for your class, they are usually public.

A basic constructor example

Let's add a basic constructor to our example above:

```cpp
#include <iostream>

class Foo
{
private:
    int m_x {};
    int m_y {};

public:
    Foo(int x, int y) // here's our constructor function that takes two initializers
    {
        std::cout << "Foo(" << x << ", " << y << ") constructed\n";
    }

    void print() const
    {
        std::cout << "Foo(" << m_x << ", " << m_y << ")\n";
    }
};

int main()
{
    Foo foo{ 6, 7 }; // calls Foo(int, int) constructor
    foo.print();

    return 0;
}
```

This program will now compile and produce the result:

```
Foo(6, 7) constructed
Foo(0, 0)
```

When the compiler sees the definition `Foo foo{ 6, 7 }`, it looks for a matching `Foo` constructor that will accept two `int` arguments. `Foo(int, int)` is a match, so the compiler will allow the definition.

At runtime, when `foo` is instantiated, memory for `foo` is allocated, and then the `Foo(int, int)` constructor is called with parameter `x` initialized to `6` and parameter `y` initialized to `7`. The body of the constructor function then executes and prints `Foo(6, 7) constructed`.

When we call the `print()` member function, you'll note that members `m_x` and `m_y` have value 0. This is because although our `Foo(int, int)` constructor function was called, it did not actually initialize the members. We'll show how to do that in the next lesson.

Related content

We discuss the differences between using copy, direct, and list initialization to initialize objects with a constructor in lesson 14.15 -- Class initialization and copy elision (https://www.learncpp.com/cpp-tutorial/class-initialization-and-copy-elision/)[3].

Constructor implicit conversion of arguments

In lesson 10.1 -- Implicit type conversion (https://www.learncpp.com/cpp-tutorial/implicit-type-conversion/)[4], we noted that the compiler will perform implicit conversion of arguments in a function call (if needed) in order to match a function definition where the parameters are a different type:

```
void foo(int, int)
{
}

int main()
{
    foo('a', true); // will match foo(int, int)

    return 0;
}
```

This is no different for constructors: the `Foo(int, int)` constructor will match any call whose arguments are implicitly convertible to `int`:

```
class Foo
{
public:
    Foo(int x, int y)
    {
    }
};

int main()
{
    Foo foo{ 'a', true }; // will match Foo(int, int) constructor

    return 0;
}
```

Constructors should not be const

A constructor needs to be able to initialize the object being constructed -- therefore, a constructor must not be const.

```
#include <iostream>

class Something
{
private:
    int m_x{};

public:
    Something() // constructors must be non-const
    {
        m_x = 5; // okay to modify members in non-const constructor
    }

    int getX() const { return m_x; } // const
};

int main()
{
    const Something s{}; // const object, implicitly invokes (non-const) constructor

    std::cout << s.getX(); // prints 5

    return 0;
}
```

Normally a non-const member function can't be invoked on a const object. However, the C++ standard explicitly states (per class.ctor.general#5 (https://eel.is/c++draft/class.ctor.general#5)[5]) that const doesn't apply to an object under construction, and only comes into effect after the constructor ends.

Constructors vs setters

Constructors are designed to initialize an entire object at the point of instantiation. Setters are designed to assign a value to a single member of an existing object.

Next lesson
14.10 Constructor member initializer lists

6

Back to table of contents

7

Previous lesson
14.8 The benefits of data hiding (encapsulation)

8

9

jack
January 18, 2025 11:05 am PST

I just want to say thanks for this unbelievably fantastic and generous resource you have made here, Alex.

29 Reply

Frank
January 14, 2025 8:26 pm PST

.

Last edited 5 months ago by Frank

3 Reply

Teretana
December 21, 2024 2:38 pm PST

"Constructors should not be const"

"Normally a non-const member function can’t be invoked on a const object. However, because the constructor is invoked implicitly, a non-const constructor can be invoked on a const object."

If i understood this correctly, the way this works is:

1. Constructor finds matching object and allocates memory for it (as a sub function assigns value of 5 to s).

At this point nothing is constant AND happens before s.getX() is called?

2. s.getX() gets called and returns value of m_x as a constant.

Does this happen because m_x allready has a value assigned by constructor ?
Const value returned is only due to getX() having a const as a return value, however this means that it actually returns a copy of the object? Or copy return is irrelevant here and i m just overthinking?

3. Because const Something s{} is a constant, getX has to be constant as well, and the constructor behavior happens because its a constructor thing (And we only use them for instantiating a variable, so we cant use them later on for editing the const value) ?

Overall everything you present is very understandable, i m just having trouble understanding relationship and priority between elements. But i love it

Thanks in advance, I started this course back in late October and am a COMPLETE begginer in programing so my questions are very basic i m sure.

Last edited 6 months ago by Teretana

0 Reply

Alex Author
Reply to Teretana 13 December 29, 2024 11:47 pm PST

Constructors don't allocate memory for the object -- they are called to initialize an object. So when `s` is being constructed, it is not treated as const. After the constructor ends, it is treated as const.

`getX()` doesn't have a const return value -- this const makes it a const member function, so that it can be called on const objects. It returns a copy of member `m_x`.

1 Reply

Teretana
Reply to Alex 14 December 31, 2024 4:45 pm PST

Thanks a lot!

0 Reply

RRR
September 15, 2024 2:24 am PDT

> That said, it's not uncommon for programmers to use default initialization instead of value initialization for class types. This is partly for historic reasons (as value initialization wasn't introduced until C++11), and partly because there is a similar case (for non-aggregates) where default initialization can be more efficient (we cover this case in 14.9 -- Introduction to constructors).

Is this mentioned?

0 Reply

Alex Author
Reply to RRR 15 September 16, 2024 4:16 pm PDT

It looks like the case was dropped when the lesson was refactored. It's been re-added to https://www.learncpp.com/cpp-tutorial/default-constructors-and-default-arguments/, and the link in the prior lesson has been updated accordingly.

Thanks for pointing this out!

1 Reply

Kevin
August 8, 2024 11:32 pm PDT

had better add this link https://www.learncpp.com/cpp-tutorial/introduction-to-stdvector-and-list-constructors/ to this article to emphasize the difference between direct initialization and list initialization in the object of class type

Last edited 10 months ago by Kevin

0 Reply

Alex Author
Reply to Kevin 16 August 10, 2024 5:10 pm PDT

We haven't even discussed that you can use different forms of initialization with constructors, or that constructors can be overloaded yet. https://www.learncpp.com/cpp-tutorial/class-initialization-and-copy-elision/ feels like a better place for this.

2 Reply

Strain
January 19, 2024 1:10 pm PST

There are a few problems with C++'s way of constructors. Most of them were covered by Logan Smith (YouTube), but one of the most significant ones is the ordering of the initialised members that can screw up things like type size. Anyway, I don't have much time, so go check out that video.

2 Reply

Inkl1ng
January 8, 2024 5:54 pm PST

I've seen constructors being called many different ways.

```
Foo foo(1);
Foo foo = Foo(1);
Foo foo {1};
```

Is there a "best practice" way or is there no difference between these?

3 Reply

Teretana
Reply to Inkl1ng 17 March 27, 2025 8:06 pm PDT

How do you "Call a constructor" ?

Is the code above from int main () or the body of the Class/Type?

0 Reply

Alex Author
Reply to Inkl1ng 17 January 9, 2024 7:11 pm PST

The third one is the best practice (with one exception for container types, noted in lesson https://www.learncpp.com/cpp-tutorial/introduction-to-stdvector-and-list-constructors/)

10 Reply

ArminC
December 24, 2023 2:23 am PST

Can you put inside the lesson, a little emphasis on the difference between setters and (parameterized) constructors? I'm not sure if it is necessary, though I find them in "same area of work" but with some key differences.

1 Reply

Alex Author

Reply to ArminC [18] December 26, 2023 6:54 pm PST

Added "Constructors are designed to initialize an entire object at the point of instantiation. Setters are designed to assign a value to a single member of an existing object."

2 Reply

Erad

Reply to Alex [19] December 20, 2024 10:18 pm PST

Are setters confined to assigning ONLY a value though? Can we have a scenario like the below?:

```
class Foo{

int m_x {};
int m_y {};

public:
  Foo(int x, int y){
    m_x = x;
    m_y = y;
  }

  //here, this setter is setting 2 variables not just one! ISN'T THIS
ALLOWED??
  void setVariables (int a, int b){
    m_x = a;
    m_y = b;
  }

};
```

Last edited 6 months ago by Erad

1 Reply

Alex Author

Reply to Erad [20] December 29, 2024 5:40 pm PST

Conventionally, getters and setters reference a single data member, are named after that member, and either assign or return that member directly.

However, in complex cases, a single getter or setter might reference or alter multiple members internally.

There is no precise definition for "setter", so whether you want to call a multi-argument setter-like function a setter or not is a matter of opinion. For something like `setXY(int x, int y)`, where m_x and m_y are members, I'd probably call that a setter. For something like `setPosition(int x, int y)` that sets m_point, I'd probably not call that a setter.

But it doesn't really matter, since this is just a word we use to describe a certain type of a function, not something for which there are any language rules around.

0 Reply

NordicCat

Reply to Erad 20 December 21, 2024 7:08 pm PST

I think it depends on what you are trying to achieve but for most of the time I have seen a separation between individual setters.

It is a nice doubt! I think both are trying to do the same thing but one is trying to do at the time when we instantiate an object while other when we want to change and assign a new value to it. I mean what's the point of doing a same thing twice?

0 Reply

Erad

Reply to NordicCat 21 December 23, 2024 6:40 pm PST

It seems like you misinterpreted my query. I'm just puzzled by the claim made by Alex in his response to the OP above:

"... Setters are designed to assign a value to a single member of an existing object."

That's why I created member function

`setVariables(int, int)`

which clearly is setting TWO variables as opposed to just ONE (single member) as stated by Alex. My question remains unanswered still!

Last edited 6 months ago by Erad

0 Reply

NordicCat

Reply to Erad 22 December 24, 2024 12:26 am PST

"It is funny that even if we know everything about OOP and c++, you are just going to use only those features that suits your program or the problem you are trying to solve. Don't sweat on small stuffs, just write program and search what suits your problem" - 'My professor said that'.

Alex wrote an amazing chapter :
https://www.learncpp.com/cpp-tutorial/using-a-language-reference/

I guarantee that 99% of your time, you are going to use c++ reference as your guide for creating program .

0 Reply

NordicCat

Reply to Erad 22 December 24, 2024 12:18 am PST ohh! sorry for that.

0 Reply

cpp_coder
October 26, 2023 6:54 am PDT

Does braced initialization NOT disallow narrowing in case of constructor?
This code compiled without error:

```
class Foo
{
public:
Foo(int x, int y)
{
}
};

int main()
{
Foo foo{2.3, true};
return 0;
}
```

0 Reply

Alex Author
Reply to cpp_coder 23 October 26, 2023 1:47 pm PDT

Doesn't compile without error for me on GCC or Clang.

1 Reply

CppLeaner
October 4, 2023 3:26 pm PDT

In the lesson, you mentioned that "A constructor is a special member function that is automatically called after a non-aggregate class type object is created."

If I understand it correctly, when we define a class variable(e.g. Employee a{"name"};) or create a temporary class object(e.g. Employee("name");), we are not explicitly calling the constructor, but instead telling the compiler to invoke the appropriate constructors implicitly for us. Is this understanding correct?

1 Reply

Alex Author
Reply to CppLeaner 24 October 5, 2023 4:58 pm PDT

Yep.

3 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/struct-aggregate-initialization/
3. https://www.learncpp.com/cpp-tutorial/class-initialization-and-copy-elision/
4. https://www.learncpp.com/cpp-tutorial/implicit-type-conversion/
5. https://eel.is/c++draft/class.ctor.general#5
6. https://www.learncpp.com/cpp-tutorial/constructor-member-initializer-lists/
7. https://www.learncpp.com/
8. https://www.learncpp.com/cpp-tutorial/the-benefits-of-data-hiding-encapsulation/
9. https://www.learncpp.com/introduction-to-constructors/
10. https://www.learncpp.com/cpp-tutorial/access-functions/
11. https://www.learncpp.com/cpp-tutorial/destructors/
12. https://gravatar.com/
13. https://www.learncpp.com/cpp-tutorial/introduction-to-constructors/#comment-605552
14. https://www.learncpp.com/cpp-tutorial/introduction-to-constructors/#comment-605940
15. https://www.learncpp.com/cpp-tutorial/introduction-to-constructors/#comment-601984
16. https://www.learncpp.com/cpp-tutorial/introduction-to-constructors/#comment-600660
17. https://www.learncpp.com/cpp-tutorial/introduction-to-constructors/#comment-592005
18. https://www.learncpp.com/cpp-tutorial/introduction-to-constructors/#comment-591315
19. https://www.learncpp.com/cpp-tutorial/introduction-to-constructors/#comment-591362
20. https://www.learncpp.com/cpp-tutorial/introduction-to-constructors/#comment-605509
21. https://www.learncpp.com/cpp-tutorial/introduction-to-constructors/#comment-605563
22. https://www.learncpp.com/cpp-tutorial/introduction-to-constructors/#comment-605643
23. https://www.learncpp.com/cpp-tutorial/introduction-to-constructors/#comment-589143
24. https://www.learncpp.com/cpp-tutorial/introduction-to-constructors/#comment-588231
25. https://g.ezoic.net/privacy/learncpp.com