# 14.11 — Default constructors and default arguments

ALEX[1] SEPTEMBER 16, 2024

A **default constructor** is a constructor that accepts no arguments. Typically, this is a constructor that has been defined with no parameters.

Here is an example of a class that has a default constructor:

```
#include <iostream>

class Foo
{
public:
    Foo() // default constructor
    {
        std::cout << "Foo default constructed\n";
    }
};

int main()
{
    Foo foo{}; // No initialization values, calls Foo's default constructor

    return 0;
}
```

When the above program runs, an object of type `Foo` is created. Since no initialization values have been provided, the default constructor `Foo()` is called, which prints:

```
Foo default constructed
```

## Value initialization vs default initialization for class types

If a class type has a default constructor, both value initialization and default initialization will call the default constructor. Thus, for such a class such as the `Foo` class in the example above, the following are essentially equivalent:

```
Foo foo{}; // value initialization, calls Foo() default constructor
Foo foo2;  // default initialization, calls Foo() default constructor
```

However, as we already covered in lesson 13.9 -- Default member initialization (https://www.learncpp.com/cpp-tutorial/default-member-initialization/)[2], value initialization is safer for aggregates. Since it's difficult to tell whether a class type is an aggregate or non-aggregate, it's safer to just use value initialization for everything and not worry about it.

### Best practice

Prefer value initialization over default initialization for all class types.

## Constructors with default arguments

As with all functions, the rightmost parameters of constructors can have default arguments.

### Related content

We cover default arguments in lesson 11.5 -- Default arguments (https://www.learncpp.com/cpp-tutorial/default-arguments/)[3].

For example:

```
#include <iostream>

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

public:
    Foo(int x=0, int y=0) // has default arguments
        : m_x { x }
        , m_y { y }
    {
        std::cout << "Foo(" << m_x << ", " << m_y << ") constructed\n";
    }
};

int main()
{
    Foo foo1{};     // calls Foo(int, int) constructor using default arguments
    Foo foo2{6, 7}; // calls Foo(int, int) constructor

    return 0;
}
```

This prints:

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

If all of the parameters in a constructor have default arguments, the constructor is a default constructor (because it can be called with no arguments).

We'll see examples of where this can be useful in the next lesson (14.12 -- Delegating constructors (https://www.learncpp.com/cpp-tutorial/delegating-constructors/)[4]).

## Overloaded constructors

Because constructors are functions, they can be overloaded. That is, we can have multiple constructors so that we can construct objects in different ways:

```cpp
#include <iostream>

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

public:
    Foo() // default constructor
    {
        std::cout << "Foo constructed\n";
    }

    Foo(int x, int y) // non-default constructor
        : m_x { x }, m_y { y }
    {
        std::cout << "Foo(" << m_x << ", " << m_y << ") constructed\n";
    }
};

int main()
{
    Foo foo1{};     // Calls Foo() constructor
    Foo foo2{6, 7}; // Calls Foo(int, int) constructor

    return 0;
}
```

A corollary of the above is that a class should only have one default constructor. If more than one default constructor is provided, the compiler will be unable to disambiguate which should be used:

```cpp
#include <iostream>

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

public:
    Foo() // default constructor
    {
        std::cout << "Foo constructed\n";
    }

    Foo(int x=1, int y=2) // default constructor
        : m_x { x }, m_y { y }
    {
        std::cout << "Foo(" << m_x << ", " << m_y << ") constructed\n";
    }
};

int main()
{
    Foo foo{}; // compile error: ambiguous constructor function call

    return 0;
}
```

In the above example, we instantiate `foo` with no arguments, so the compiler will look for a default constructor. It will find two, and be unable to disambiguate which constructor should be used. This will result in a compile error.

## An implicit default constructor

If a non-aggregate class type object has no user-declared constructors, the compiler will generate a public default constructor (so that the class can be value or default initialized). This constructor is called an **implicit default constructor**.

Consider the following example:

```
#include <iostream>

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

    // Note: no constructors declared
};

int main()
{
    Foo foo{};

    return 0;
}
```

This class has no user-declared constructors, so the compiler will generate an implicit default constructor for us. That constructor will be used to instantiate `foo{}`.

The implicit default constructor is equivalent to a constructor that has no parameters, no member initializer list, and no statements in the body of the constructor. In other words, for the above `Foo` class, the compiler generates this:

```
public:
    Foo() // implicitly generated default constructor
    {
    }
```

The implicit default constructor is useful mostly when we have classes that have no data members. If a class has data members, we'll probably want to make them initializable with values provided by the user, and the implicit default constructor isn't sufficient for that.

---

## Using `= default` to generate an explicitly defaulted default constructor

In cases where we would write a default constructor that is equivalent to the implicitly generated default constructor, we can instead tell the compiler to generate a default constructor for us. This constructor is called an **explicitly defaulted default constructor**, and it can be generated by using the `= default` syntax:

```cpp
#include <iostream>

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

public:
    Foo() = default; // generates an explicitly defaulted default constructor

    Foo(int x, int y)
        : m_x { x }, m_y { y }
    {
        std::cout << "Foo(" << m_x << ", " << m_y << ") constructed\n";
    }
};

int main()
{
    Foo foo{}; // calls Foo() default constructor

    return 0;
}
```

In the above example, since we have a user-declared constructor (`Foo(int, int)`), an implicit default constructor would not normally be generated. However, because we've told the compiler to generate such a constructor, it will. This constructor will subsequently be used by our instantiation of `foo{}`.

### Best practice

Prefer an explicitly defaulted default constructor (`= default`) over a default constructor with an empty body.

---

## Explicitly defaulted default constructor vs empty user-defined constructor

There are at least two cases where the explicitly defaulted default constructor behaves differently than an empty user-defined constructor.

1. When value initializing a class, if the class has a user-defined default constructor, the object will be default initialized. However, if the class has a default constructor that is not user-provided (that is, a default constructor that is either implicitly defined, or defined using `= default`), the object will be zero-initialized before being default initialized.

```cpp
#include <iostream>

class User
{
private:
    int m_a; // note: no default initialization value
    int m_b {};

public:
    User() {} // user-defined empty constructor

    int a() const { return m_a; }
    int b() const { return m_b; }
};

class Default
{
private:
    int m_a; // note: no default initialization value
    int m_b {};

public:
    Default() = default; // explicitly defaulted default constructor

    int a() const { return m_a; }
    int b() const { return m_b; }
};

class Implicit
{
private:
    int m_a; // note: no default initialization value
    int m_b {};

public:
    // implicit default constructor

    int a() const { return m_a; }
    int b() const { return m_b; }
};

int main()
{
    User user{}; // default initialized
    std::cout << user.a() << ' ' << user.b() << '\n';

    Default def{}; // zero initialized, then default initialized
    std::cout << def.a() << ' ' << def.b() << '\n';

    Implicit imp{}; // zero initialized, then default initialized
    std::cout << imp.a() << ' ' << imp.b() << '\n';

    return 0;
}
```

On the author's machine, this prints:

```
782510864 0
0 0
0 0
```

Note that `user.a` was not zero initialized before being default initialized, and thus was left uninitialized.

In practice, this shouldn't matter since you should be providing default member initializers for all data members!

> **Tip**
>
> For a class that does not have a user-provided default constructor, value initialization will first zero-initialize the class, while default initialization will not. Given this, default initialization may be more performant than value initialization (at the cost of being less safe). If you are looking to squeeze out every bit of performance in a section of code that initializes a lot of objects that do not have user-provided default constructors, changing those objects to be default initialized may be worth exploring. Alternatively, you could try changing the class to have a default constructor with an empty body. This avoids the zero-initialization case when using value initialization, but may inhibit other optimizations.

2. Prior to C++20, a class with a user-defined default constructor (even if it has an empty body) makes the class a non-aggregate, whereas an explicitly defaulted default constructor does not. Assuming the class was otherwise an aggregate, the former would cause the class to use list initialization instead of aggregate initialization. In C++20 onward, this inconsistency was addressed, so that both make the class a non-aggregate.

---

## Only create a default constructor when it makes sense

A default constructor allows us to create objects of a non-aggregate class type with no user-provided initialization values. Thus, a class should only provide a default constructor when it makes sense for objects of a class type to be created using all default values.

For example:

```cpp
#include <iostream>

class Fraction
{
private:
    int m_numerator{ 0 };
    int m_denominator{ 1 };

public:
    Fraction() = default;
    Fraction(int numerator, int denominator)
        : m_numerator{ numerator }
        , m_denominator{ denominator }
    {
    }

    void print() const
    {
        std::cout << "Fraction(" << m_numerator << ", " << m_denominator << ")\n";
    }
};

int main()
{
    Fraction f1 {3, 5};
    f1.print();

    Fraction f2 {}; // will get Fraction 0/1
    f2.print();

    return 0;
}
```

For a class representing a fraction, it makes sense to allow the user to create Fraction objects with no initializers (in which case, the user will get the fraction 0/1).

Now consider this class:

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

class Employee
{
private:
    std::string m_name{ };
    int m_id{ };

public:
    Employee(std::string_view name, int id)
        : m_name{ name }
        , m_id{ id }
    {
    }

    void print() const
    {
        std::cout << "Employee(" << m_name << ", " << m_id << ")\n";
    }
};

int main()
{
    Employee e1 { "Joe", 1 };
    e1.print();

    Employee e2 {}; // compile error: no matching constructor
    e2.print();

    return 0;
}
```

For a class representing an employee, it doesn't make sense to allow creation of employees with no name. Thus, such a class should not have a default constructor, so that a compilation error will result if the user of the class tries to do so.


**Next lesson**

14.12 Delegating constructors

4

**Back to table of contents**

5

**Previous lesson**

14.10 Constructor member initializer lists

6

7

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/[10] are connected to your provided email address.

Notify me about replies:

POST COMMENT

## 51 COMMENTS

Newest

### Felipe

June 22, 2025 1:48 pm PDT

I don't understand how an object can be zero initialized and default initialized. I thought initialization happened only once per object.

Also, what exactly is default initialization?

0 Reply

### Nidhi Gupta

March 16, 2025 9:35 pm PDT

Separating the class declarations from the implementations has led to better organization, readability, and maintainability of the code. Small member functions may be defined in the class declaration; however, complex functions should be implemented outside it in order not to clutter the interface. With C++, class declarations can be defined in header (.h) files and implementations in source (.cpp) files, allowing reuse of code across multiple files. Header guards help to avoid the problem of multiple inclusion errors during compilation. Inline functions, such as those that are defined inside the class, work to prevent ODR violations. Although one can define all functions in a header file, building all functions in one header file might prolong compilation time. Therefore, it is better to place non-trivial functions in a source file. That also makes extensibility easier via precompiled libraries, giving a better edge to large-scale software development.

1 Reply

### JiaChen

March 1, 2025 11:30 pm PST

A note could be added to compare with the following example from 13.9. The struct's x would be initialized also because for an aggreate there is an implicit default constructor from my understanding.

```
struct Something
{
    int x;       // no default initialization value (bad)
    int y {};    // value-initialized by default
    int z { 2 }; // explicit default value
};

int main()
{
    Something s3 {}; // value initialize s3.x, use default values for s3.y and s3.z

    return 0;
}
```

1 Reply

NordicCat

December 22, 2024 1:05 am PST

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


class Weapon {
public :
    // default constructor
    Weapon() = default;
    // Or there is a hybrid version of default constructor
    Weapon() {
        weight = 1.0;
        recoilRate = 0.1;
        damage = 5.0;
        numOfBullets = 25;
        totalBullets = 60;
    }
private:
    // Do not left them Unitialised untill you have a specific reason to do so
    float weight{};
    float recoilRate{};
    float damage{};
    int numOfBullets{};
    int totalBullets{};
};

int main() {
    Weapon phantom{}; // let compliler give default values to specific data types like
int = 0, float = 0.0, etc.

    // Or We Use Default Constructor that we created explicitely (I prefer this, btw
Idk the cost)
    Weapon vandal{};
}
```

Which should we prefer Alex?

1 Reply

Nidhi Gupta

Reply to NordicCat [11] March 16, 2025 9:33 pm PDT

cool

0 Reply

Alex Author

Reply to NordicCat [11] December 30, 2024 12:21 am PST

Unless you have a specific reason to do otherwise, I'd probably choose this:

```
class Weapon
{
public :
    // Make sure we always have a default constructor even if we define other
constructors later
    Weapon() = default;

private:
        // Everything has a reasonable default value
    float weight{ 1.0f };
    float recoilRate{ 0.1f };
    float damage{ 5.0f };
    int numOfBullets{ 25 };
    int totalBullets{ 60 };
};
```

0 Reply

NordicCat

Reply to Alex [12] December 30, 2024 8:08 am PST

So this means when we want custom default values assigned to our member variables we should member initialize the variables + set our intent explicit by making a explicit default constructor by using (=default). Wouldn't it implicitly generate a default constructor if we do it like this?

```
private:
        // Everything has a reasonable default value
    float weight{ 1.0f };
    float recoilRate{ 0.1f };
    float damage{ 5.0f };
    int numOfBullets{ 25 };
    int totalBullets{ 60 };
};
```

Isn't it redundant then (i mean =default)?

0 Reply

Alex Author

Reply to NordicCat [13] January 4, 2025 4:07 pm PST

Yes, the =default constructor is superfluous and can be left out if you want. Like the comment says, it's there to make it clear you intend for objects to be default constructed, even if another constructor is added later.

0 Reply

**Cpp Learner**

December 13, 2024 7:01 pm PST

I feeling like if I don't need a constructor, that means probably my requirement is aggregate. Since classes are meant to be for non-aggregate complex types and aggregates are for simple basic requirements, In this case if I don't need a constructor I should definitely use structs?

1 Reply

**Alex** Author

Reply to Cpp Learner [14] December 19, 2024 9:46 pm PST

Not quite.

If your requirement is an aggregate, then use a struct (and no constructor).

If you need your data to be private/encapsulated, then use a class. Using access controls will make the class a non-aggregate. That class doesn't have to have a user-defined constructor -- in which case it will be given an implicit default constructor. That might be fine, depending on your use case.

0 Reply

**Sam P Adams**

December 6, 2024 11:39 am PST

Is there a preference between using an implicit default constructor vs explicitly defaulted default constructor?

0 Reply

**Alex** Author

Reply to Sam P Adams [15] December 9, 2024 9:18 pm PST

You only get an implicit default constructor when you have no other constructors. In that case, I don't see a strong reason to prefer an explicitly defaulted default constructor. One potential pitfall of sticking with the implicit default constructor is that if you add a constructor later, you will no longer get an implicit default constructor, and any code that uses the implicit default constructor will not compile until you define one. But I don't think it's worth favoring an implicit default constructor to address this case.

You only need an explicitly default default constructor when you have a constructor but also want a default constructor that would behave identically to the implicit default constructor. In this case the implicit default constructor won't even be generated, so you have to explicitly define a default constructor or use an explicitly defaulted default constructor (preferred).

0 Reply

**Seeminglyimplicitexplicitconversion**
November 9, 2024 3:32 am PST

Might want to mention that if the default constructor can be constexpr, it will be automatically made constexpr.
As per: https://en.cppreference.com/w/cpp/language/default_constructor

"If this satisfies the requirements of a constexpr constructor(until C++23)constexpr function(since C++23), the generated constructor is constexpr. (since C++11)"

```
class Vector2
{
public:
    Vector2() = default;
    Vector2(double x, double y) : m_x{ x }, m_y{ y } {} // not constexpr

private:
    double m_x{ 0.0 };
    double m_y{ 0.0 };
};

int main()
{
    constexpr Vector2 vec2{}; // Works since default constructor is automatically made
constexpr
    constexpr Vector2 vec3{1.0, 2.0}; // Doesn't work since constructor is not
constexpr

    return 0;
}
```

*Last edited 7 months ago by Seeminglyimplicitexplicitconversion*

0 Reply

**Alex** Author
Reply to Seeminglyimplicitexplicitconversion [16] November 10, 2024 9:58 pm PST

This is mentioned in lesson https://www.learncpp.com/cpp-tutorial/constexpr-aggregates-and-classes/#google_vignette

0 Reply

**SunMan**
October 6, 2024 11:40 am PDT

Took me some time to understand this so adding notes here

/*
Prior to C++20,
Foo() {} // user-defined default constructor (even if it has an empty body)
1- Makes the class a non-aggregate

2- You can use list initialization e.g. Foo f1{};
3- You can't use aggregate initialization e.g. Foo f2 = {1,2};

Foo() = default // explicitly defaulted default constructor
1- Keeps the class an aggregate
2- List initialization (allowed) e.g. Foo f1{};
3- // Aggregate initialization (allowed) e.g. Foo f2 = {1,2} without any issues;

In C++20 onward // Fixed behavior of explicitly defaulted default constructor
1- Both make the class a non-aggregate.
2- You can use list initialization e.g. Foo f1{};
3- You can't use aggregate initialization e.g. Foo f2 = {1,2};
*/

4 Reply

**Tom**
September 21, 2024 12:57 pm PDT

Why does this code initialize m_value to zero?

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

class Data
{
public:
    Data() {};
    int m_value;
    std::string m_name;
};

int main()
{
    Data data{};
    std::cout << data.m_value ; // prints zero

    return 0;
}
```

But this one doesn't

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

class Data
{
public:
    Data() {};
    std::string m_name;
    int m_value;
};

int main()
{
    Data data{};
    std::cout << data.m_value ; // undefined behavior

    return 0;
}
```

0 Reply

**Alex** Author

Reply to Tom [17] September 23, 2024 8:27 pm PDT

m_value is uninitialized in both cases, so printing m_value is undefined behavior. Your getting 0 in one case and not the other is just happenstance.

2 Reply

**MOEGMA25**

July 15, 2024 6:08 am PDT

**Since it's difficult to tell whether a class type is an aggregate or non-aggregate, it's safer to just use value initialization for everything and not worry about it.**

Wouldn't it be an idea to just use value initialization for aggregates only? So you can tell appart structs and classes.

*Last edited 11 months ago by MOEGMA25*

0 Reply

**Alex** Author

Reply to MOEGMA25 [18] July 17, 2024 3:41 pm PDT

No. The whole point is that we don't want to have to figure out whether something is an aggregate or not in order to use it properly.

1 Reply

# Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/default-member-initialization/
3. https://www.learncpp.com/cpp-tutorial/default-arguments/
4. https://www.learncpp.com/cpp-tutorial/delegating-constructors/
5. https://www.learncpp.com/
6. https://www.learncpp.com/cpp-tutorial/constructor-member-initializer-lists/
7. https://www.learncpp.com/default-constructors-and-default-arguments/
8. https://www.learncpp.com/cpp-tutorial/the-benefits-of-data-hiding-encapsulation/
9. https://www.learncpp.com/cpp-tutorial/class-templates-with-member-functions/
10. https://gravatar.com/
11. https://www.learncpp.com/cpp-tutorial/default-constructors-and-default-arguments/#comment-605571
12. https://www.learncpp.com/cpp-tutorial/default-constructors-and-default-arguments/#comment-605951
13. https://www.learncpp.com/cpp-tutorial/default-constructors-and-default-arguments/#comment-605966
14. https://www.learncpp.com/cpp-tutorial/default-constructors-and-default-arguments/#comment-605221
15. https://www.learncpp.com/cpp-tutorial/default-constructors-and-default-arguments/#comment-604977
16. https://www.learncpp.com/cpp-tutorial/default-constructors-and-default-arguments/#comment-603973
17. https://www.learncpp.com/cpp-tutorial/default-constructors-and-default-arguments/#comment-602189
18. https://www.learncpp.com/cpp-tutorial/default-constructors-and-default-arguments/#comment-599615
19. https://g.ezoic.net/privacy/learncpp.com