14.11 — Default constructors and default arguments

ALEX DECEMBER 26, 2023

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 13.7 -- Default member initialization (https://www.learncpp.com/cpp-tutorial/default-member-initialization/), 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/).

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/)).

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:

```
#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:

```
#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 a 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 an implicit default constructor for us by using the following syntax:

```
#include <iostream>

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

public:
    Foo() = default; // generate an implicit 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 implicitly defined 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{}`.

Implicit default constructor vs empty user constructor

There is at least one case where the implicit default constructor behaves differently than an empty user-provided constructor. When value initializing a class, if the class has a user-provided default constructor, the object will be default initialized. However, if the class has a default constructor that is not user-provided (that is, either an implicitly-defined constructor, or a default constructor created using =default), the object will be zero-initialized before being default initialized.

```
#include <iostream>

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

public:
    User() {} // user-provided 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!

Best practice

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

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:

```
#include <iostream>

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

public:
    Fraction() {}
    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:

```
#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

Back to table of contents

Previous lesson
14.10 Constructor member initializer lists