14.12 — Delegating constructors

ALEX[1] JANUARY 21, 2025

Whenever possible, we want to reduce redundant code (following the DRY principle -- Don't Repeat Yourself).

Consider the following functions:

```
void A()
{
    // statements that do task A
}

void B()
{
    // statements that do task A
    // statements that do task B
}
```

Both functions have a set of statements that do exactly the same thing (task A). In such a case, we can refactor like this:

```
void A()
{
    // statements that do task A
}

void B()
{
    A();
    // statements that do task B
}
```

In that way, we've removed redundant code that existed in functions `A()` and `B()`. This makes our code easier to maintain, as changes only need to be made in one place.

When a class contains multiple constructors, it's extremely common for the code in each constructor to be similar if not identical, with lots of repetition. We'd similarly like to remove constructor redundancy where possible.

Consider the following example:

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

class Employee
{
private:
    std::string m_name { "???" };
    int m_id { 0 };
    bool m_isManager { false };

public:
    Employee(std::string_view name, int id) // Employees must have a name and an id
        : m_name{ name }, m_id { id }
    {
        std::cout << "Employee " << m_name << " created\n";
    }

    Employee(std::string_view name, int id, bool isManager) // They can optionally be
a manager
        : m_name{ name }, m_id{ id }, m_isManager { isManager }
    {
        std::cout << "Employee " << m_name << " created\n";
    }
};

int main()
{
    Employee e1{ "James", 7 };
    Employee e2{ "Dave", 42, true };
}
```

The body of each constructor has the exact same print statement.

Author's note

It's generally not a good idea to have a constructor print something (except for debugging purposes), as this means you can't create an object using that constructor in cases where you do not want to print something. We're doing it in this example to help illustrate what's happening.

Constructors are allowed to call other functions, including other member functions of the class. So we could refactor like this:

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

class Employee
{
private:
    std::string m_name { "???" };
    int m_id{ 0 };
    bool m_isManager { false };

    void printCreated() const // our new helper function
    {
        std::cout << "Employee " << m_name << " created\n";
    }

public:
    Employee(std::string_view name, int id)
        : m_name{ name }, m_id { id }
    {
        printCreated(); // we call it here
    }

    Employee(std::string_view name, int id, bool isManager)
        : m_name{ name }, m_id{ id }, m_isManager { isManager }
    {
        printCreated(); // and here
    }
};

int main()
{
    Employee e1{ "James", 7 };
    Employee e2{ "Dave", 42, true };
}
```

While this is better than the prior version (as the redundant statement has been replaced by a redundant function call), it requires introduction of a new function. And our two constructors are also both initializing `m_name` and `m_id`. Ideally we'd remove this redundancy too.

Can we do better? We can. But this is where many new programmers run into trouble.

Calling a constructor in the body of a function creates a temporary object

Analogous to how we had function `B()` call function `A()` in the example above, the obvious solution seems like it would be to call the `Employee(std::string_view, int)` constructor from the body of `Employee(std::string_view, int, bool)` in order to initialize `m_name`, `m_id`, and print the statement. Here's what that looks like:

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

class Employee
{
private:
    std::string m_name { "???" };
    int m_id { 0 };
    bool m_isManager { false };

public:
    Employee(std::string_view name, int id)
        : m_name{ name }, m_id { id } // this constructor initializes name and id
    {
        std::cout << "Employee " << m_name << " created\n"; // our print statement is
back here
    }

    Employee(std::string_view name, int id, bool isManager)
        : m_isManager { isManager } // this constructor initializes m_isManager
    {
        // Call Employee(std::string_view, int) to initialize m_name and m_id
        Employee(name, id); // this doesn't work as expected!
    }

    const std::string& getName() const { return m_name; }
};

int main()
{
    Employee e2{ "Dave", 42, true };
    std::cout << "e2 has name: " << e2.getName() << "\n"; // print e2.m_name
}
```

But this doesn't work correctly, as the program outputs the following:

```
Employee Dave created
e2 has name: ???
```

Even though `Employee Dave created` printed, after `e2` finished construction, `e2.m_name` still appears to be set to its initial value of `"???"`. How is that possible?

We were expecting `Employee(name, id)` to call the constructor in order to continue initialization of the current implicit object (`e2`). But initialization of a class object is finished once the member initializer list has finished executing. By the time we begin executing the constructor's body, it's too late to do more initialization.

When called from the body of a function, what looks like a function call to a constructor usually creates and direct-initializes a temporary object (in one other case, you'll get a compile error instead). In our example above, `Employee(name, id);` creates a temporary (unnamed) Employee object. This temporary object is the one whose `m_name` is set to `Dave`, and is the one that prints `Employee Dave created`. Then the temporary is destroyed. `e2` never has its `m_name` or `m_id` changed from the default values.

Best practice

Constructors should not be called directly from the body of another function. Doing so will either result in a compilation error, or will direct-initialize a temporary object.

If you do want a temporary object, prefer list-initialization (which makes it clear you are intending to create an object).

So if we can't call a constructor from the body of another constructor, then how do we solve this issue?

Delegating constructors

Constructors are allowed to delegate (transfer responsibility for) initialization to another constructor from the same class type. This process is sometimes called **constructor chaining** and such constructors are called **delegating constructors**.

To make one constructor delegate initialization to another constructor, simply call the constructor in the member initializer list:

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

class Employee
{
private:
    std::string m_name { "???" };
    int m_id { 0 };

public:
    Employee(std::string_view name)
        : Employee{ name, 0 } // delegate initialization to Employee(std::string_view,
int) constructor
    {
    }

    Employee(std::string_view name, int id)
        : m_name{ name }, m_id { id } // actually initializes the members
    {
        std::cout << "Employee " << m_name << " created\n";
    }

};

int main()
{
    Employee e1{ "James" };
    Employee e2{ "Dave", 42 };
}
```

When `e1 { "James" }` is initialized, matching constructor `Employee(std::string_view)` is called with parameter `name` set to `"James"`. The member initializer list of this constructor delegates initialization to other constructor, so `Employee(std::string_view, int)` is then called. The value of `name` (`"James"`) is passed as the first argument, and literal `0` is passed as the second argument. The member initializer list of the delegated constructor then initializes the members. The body of the delegated constructor then runs. Then control returns to the initial constructor, whose (empty) body runs. Finally, control returns to the caller.

The downside of this method is that it sometimes requires duplication of initialization values. In the delegation to the `Employee(std::string_view, int)` constructor, we need an initialization value for the `int` parameter. We had to hardcode literal `0`, as there is no way to reference the default member initializer.

A few additional notes about delegating constructors. First, a constructor that delegates to another constructor is not allowed to do any member initialization itself. So your constructors can delegate or initialize, but not both.

As an aside...

Note that we had `Employee(std::string_view)` (the constructor with less parameters) delegate to `Employee(std::string_view name, int id)` (the constructor with more parameters). It is common to have the constructor with fewer parameters delegate to the constructor with more parameters.

If we had instead chosen to have `Employee(std::string_view name, int id)` delegate to `Employee(std::string_view)`, then that would have left us unable to initialize `m_id` using `id`, as a constructor can only delegate or initialize, not both.

Second, it's possible for one constructor to delegate to another constructor, which delegates back to the first constructor. This forms an infinite loop, and will cause your program to run out of stack space and crash. You can avoid this by ensuring all of your constructors resolve to a non-delegating constructor.

Best practice

If you have multiple constructors, consider whether you can use delegating constructors to reduce duplicate code.

Reducing constructors using default arguments

Default values can also sometimes be used to reduce multiple constructors into fewer constructors. For example, by putting a default value on our `id` parameter, we can create a single `Employee` constructor that requires a name argument but will optionally accept an id argument:

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

class Employee
{
private:
    std::string m_name{};
    int m_id{ 0 }; // default member initializer

public:

    Employee(std::string_view name, int id = 0) // default argument for id
        : m_name{ name }, m_id{ id }
    {
        std::cout << "Employee " << m_name << " created\n";
    }
};

int main()
{
    Employee e1{ "James" };
    Employee e2{ "Dave", 42 };
}
```

Since default values must be attached to the rightmost parameters in a function call, a good practice when defining classes is to define members for which a user *must* provide initialization values for first (and then make those the leftmost parameters of the constructor). Members for which the user can optionally provide (because the default values are acceptable) should be defined second (and then make those the rightmost parameters of the constructor).

Best practice

Members for which the user must provide initialization values should be defined first (and as the leftmost parameters of the constructor). Members for which the user can optionally provide initialization values (because the default values are acceptable) should be defined second (and as the rightmost parameters of the constructor).

Note that this method also requires duplication of the default initialization value for `m_id` ('0'): once as a default member initializer, and once as a default argument.

A conundrum: Redundant constructors vs redundant default values

In the above examples, we used delegating constructors and then default arguments to reduce constructor redundancy. But both of these methods required us to duplicate initialization values for our members in various places. Unfortunately, there is currently no way to specify that a delegating constructor or default argument should use the default member initializer value.

There are various opinions about whether it is better to have fewer constructors (with duplication of initialization values) or more constructors (with no duplication of initialization values). Our opinion is that it's usually more straightforward to have fewer constructors, even if it results in duplication of initialization values.

For advanced readers

When we have an initialization value that is used in multiple places (e.g. as a default member initializer and a default argument for a constructor parameter), we can define a named constant and use that wherever our initialization value is needed. This allows the initialization value to be defined in one place.

Although you could use a constexpr global variable for this, a better option is to use a `static constexpr` member inside the class:

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

class Employee
{
private:
    static constexpr int default_id { 0 }; // define a named constant with our
desired initialization value

    std::string m_name {};
    int m_id { default_id }; // we can use it here

public:

    Employee(std::string_view name, int id = default_id) // and we can use it here
        : m_name { name }, m_id { id }
    {
        std::cout << "Employee " << m_name << " created\n";
    }
};

int main()
{
    Employee e1 { "James" };
    Employee e2 { "Dave", 42 };
}
```

Use of the `static` keyword in this context allows us to have a single `default_id` member that is shared by all `Employee` objects. Without the `static`, each `Employee` object would have its own independent `default_id` member (which would work, but be a waste of memory).

The downside of this approach is that each additional named constant adds another name that must be understood, making your class a little more cluttered and complex. Whether this is worth it depends on how many of such constants are required, and in how many places the initialization values are needed.

We cover static data members in lesson [15.6 -- Static member variables (https://www.learncpp.com/cpp-tutorial/static-member-variables/)](https://www.learncpp.com/cpp-tutorial/static-member-variables/)[2].

Quiz time

Question #1

Write a class named Ball. Ball should have two private member variables, one to hold a color (default value: `black`), and one to hold a radius (default value: `10.0`). Add 4 constructors, one to handle each case below:

```cpp
int main()
{
    Ball def{};
    Ball blue{ "blue" };
    Ball twenty{ 20.0 };
    Ball blueTwenty{ "blue", 20.0 };

    return 0;
}
```

The program should produce the following result:

```
Ball(black, 10)
Ball(blue, 10)
Ball(black, 20)
Ball(blue, 20)
```

Show Solution (javascript:void(0))[3]

Question #2

Reduce the number of constructors in the above program by using default arguments and delegating constructors.

Show Solution (javascript:void(0))[3]

Next lesson
14.13 Temporary class objects

4

Back to table of contents

5

Previous lesson
14.11 Default constructors and default arguments

6

7

Lin

June 11, 2025 10:15 pm PDT

"Note that this method also requires duplication of the default initialization value for m_id ('0'): once as a default member initializer, and once as a default argument." -- why do we have to specify a default member initializer?

1 Reply

Copernicus

May 29, 2025 5:58 am PDT

Question #2

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

class Ball
{
public:
    Ball(double radius)
        : Ball{"black", radius}
    {
        std::cout << "Ball(" << m_colour << ", " << m_radius << ")\n";
    }
    Ball(const std::string& colour = "black", double radius = 10.0)
        : m_colour{ colour }, m_radius{radius}
    {
        std::cout << "Ball(" << m_colour << ", " << m_radius << ")\n";
    }

private:
    std::string m_colour{ "black" };
    double m_radius{ 10.0 };
};

int main()
{
    Ball def{};
    Ball blue{ "blue" };
    Ball twenty{ 20.0 };
    Ball blueTwenty{ "blue", 20.0 };

    return 0;
}
```

0 Reply

Copernicus

May 29, 2025 5:46 am PDT

Question #1

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

class Ball
{
public:
    Ball()
    {
        std::cout << "Ball(" << m_colour << ", " << m_radius << ")\n";
    }
    Ball(const std::string& colour)
        : m_colour{ colour }
    {
        std::cout << "Ball(" << m_colour << ", " << m_radius << ")\n";
    }
    Ball(double radius)
        : m_radius{ radius }
    {
        std::cout << "Ball(" << m_colour << ", " << m_radius << ")\n";
    }
    Ball(const std::string& colour, double radius)
        : m_colour{ colour }, m_radius{radius}
    {
        std::cout << "Ball(" << m_colour << ", " << m_radius << ")\n";
    }

private:
    std::string m_colour{ "black" };
    double m_radius{ 10.0 };
};

int main()
{
    Ball def{};
    Ball blue{ "blue" };
    Ball twenty{ 20.0 };
    Ball blueTwenty{ "blue", 20.0 };

    return 0;
}
```

0 Reply

TheCodeIsBugged

May 21, 2025 3:58 am PDT

Why do the following constructors result in a compilation error or an ambiguous match?

```cpp
Ball(std::string_view color)
    : Ball(color, 10.0)
{

}

Ball(std::string_view color = "black", double radius = 10.0)
    : m_color{ color }
    , m_radius{ radius }
{
    print();
}
```

Error (active) E0309 more than one instance of constructor "Ball::Ball" matches the argument list: chapter14-classes

main.cpp 159 function "Ball::Ball(std::string_view color)" (declared at line 92)
function "Ball::Ball(std::string_view color = "black", double radius = (10.0))" (declared at line 98)
argument types are: (const char [5])

Last edited 1 month ago by TheCodeIsBugged

0 Reply

Niklas

Reply to TheCodeIsBugged [11] May 22, 2025 6:53 am PDT

Because you can call both with just a single string_view argument. Thus the compiler doesn't know which of the constructors to call.

0 Reply

TheCodeIsBugged

Reply to Niklas [12] May 30, 2025 6:32 pm PDT

Thanks, I finally understand why Ball(double radius) doesn't call the default constructor, while Ball(std::string_view color) does.

0 Reply

Zeca

May 1, 2025 12:00 pm PDT

This is my solution for the Question #2

```cpp
#include <iostream>

class Ball
{
private:
    std::string m_color{"black"};
    double m_radius{10.0};

public:
    Ball(std::string_view color = "black", double radius = 10.0)
        : m_color{color}, m_radius{radius}
    {
        print();
    }

    Ball(double radius)
        : Ball{"black", radius}
    {
    }

    void print(void) const
    {
        std::cout << "Ball" << '(' << m_color << ',' << ' ' << m_radius << ")\n";
    }
};

int main()
{
    Ball def{};
    Ball blue{ "blue" };
    Ball twenty{ 20.0 };
    Ball blueTwenty{ "blue", 20.0 };

    return 0;
}
```


Lion

April 28, 2025 5:49 pm PDT

For the first question, I wasn't able to come up with four different constructors. So I went with two...' till I saw question #2...
Anyway, I'm pretty proud with my solution.

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

class Ball
{
private:
    std::string m_color{};
    double m_radius{};

public:
    Ball(std::string_view color = "black", double radius = 10.0)
        : m_color{ color }, m_radius{ radius }
    {
        print();
    }

    Ball(double radius, std::string_view color = "black")
        : m_radius{radius}, m_color{color}
    {
        print();
    }

    void print()
    {
        std::cout << "Ball(" << m_color << ", " << m_radius << ")\n";
    }
};

int main()
{
    Ball def{};
    Ball blue{ "blue" };
    Ball twenty{ 20.0 };
    Ball blueTwenty{ "blue", 20.0 };

    return 0;
}
```

0 Reply

Jørgen

April 10, 2025 9:33 am PDT

Besides my other point I search for some direction between delegate constructors or default values. Is my thought below possibly a direction? I would like to here your vision if possible.

1. As long as it's possible to keep the number of constructor variants down to one, add default values in this single constructor.
2. Otherwise, if the demands are such complex that you need more than one constructor anyway, than make all extra constructors delegate constructors (where possible of course, otherwise separate of course).

This because I find it difficult to compare all kinds of different constructors declarations. So if possible I would prefer to see all variants in one single declaration.

0 Reply

Jørgen
April 10, 2025 9:24 am PDT

I like your course, thanks. I'm trying to develop good habits. So this comment/question about the need for redundancy.

The example for advanced readers is about a default in two places:

1. a default argument in a constructor (or a default value in the delegate constructor call with the same purpose)
2. a default member initialization

I think having the same default on two places might be a self created problem. I don't understand what we gain by having redundant:

1. default member initialization
2. initialization by the member initializer list

I would like to understand better why we would do that. This because I don't think that the default member initialization (point 2) will help us, if we forget initialization in the member initialization list. It's not more than that it's a good habit to always default initialize. If we consequently take care that we have a complete member initialization list (and we have to check the sequence anyway), that should be fine. So in my point of view we good consequently make an exception not to default member initialize in classes (structs is an other story).

So I would come to this:

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

class Employee
{
private:
std::string m_name; //no default member initialization
int m_id; //no default member initialization

public:
Employee(std::string_view name, int id = 0)
: m_name { name }, m_id { id }
{
std::cout << "Employee " << m_name << " created\n";
}
};

int main()
{
Employee e1 { "James" };
Employee e2 { "Dave", 42 };
}
```

The only thing I could imagine is that we use a sentinel (e.g. 9999) in the default member initialization for tracking purposes. With that tracking a forgotten initialization in the member initializer list would get easier. But I think that would look ugly and some objects don't have an easy sentinel so it would be incomplete anyway.

I think this way also the need for a default defined as a static constexpr would decrease. Although it could be helpful for very long class declarations to have al defaults in one place I can imagine.

0 Reply

Vishnu VK
April 1, 2025 10:05 pm PDT

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

using namespace std;

class Ball
{
private:
string m_color{"black"};
float m_radius{10.0};

public:

Ball(string& color)
:Ball{color,10.0} {}

Ball(const string& color = "black",float radius = 10.0)
: m_color(color),m_radius{radius}
{
cout<<"Ball ("<<m_color<<", "<<m_radius<<")"<<endl;

}

};

int main()
{
Ball def{};
Ball blue{ "blue" };
Ball twenty{20.0};
Ball blueTwenty{ "blue", 20.0 };

return 0;
}
```

Last edited 2 months ago by Vishnu VK

0 Reply

Vishnu VK

Reply to Vishnu VK [13] April 1, 2025 10:06 pm PDT

Why isn't this working?

0 Reply

AddText

March 5, 2025 5:26 pm PST

For Question 1, you should add "Also write a function to print out the color and radius of the ball." to the description to keep it consistent with the description for the similar program for quiz question 1 of chapter 14.10, since you are again asking to write the class named Ball, which assumes that the program doesn't already exist.

Also, the old print function could be kept and called using print(*this), which allows us to keep the function outside of the class since it doesn't need direct access if we keep the getter and setter. We would need to forward declare both the class and the function though.

Last edited 3 months ago by AddText

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/static-member-variables/
3. javascript:void(0)
4. https://www.learncpp.com/cpp-tutorial/temporary-class-objects/
5. https://www.learncpp.com/
6. https://www.learncpp.com/cpp-tutorial/default-constructors-and-default-arguments/
7. https://www.learncpp.com/delegating-constructors/
8. https://www.learncpp.com/cpp-tutorial/the-hidden-this-pointer-and-member-function-chaining/
9. https://www.learncpp.com/cpp-tutorial/classes-and-header-files/
10. https://gravatar.com/
11. https://www.learncpp.com/cpp-tutorial/delegating-constructors/#comment-610263
12. https://www.learncpp.com/cpp-tutorial/delegating-constructors/#comment-610301
13. https://www.learncpp.com/cpp-tutorial/delegating-constructors/#comment-608974
14. https://g.ezoic.net/privacy/learncpp.com