14.12 — Delegating constructors

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 };

public:
    Employee(std::string_view name)
        : m_name{ name }
    {
        std::cout << "Employee " << m_name << " created\n";
    }

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

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

The body of each constructor prints the same thing.

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 };

    void printCreated() const
    {
        std::cout << "Employee " << m_name << " created\n";
    }

public:
    Employee(std::string_view name)
        : m_name{ name }
    {
        printCreated();
    }

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

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

While this is better than the prior version, it requires introduction of a new function, which is less than ideal.

Can we do better?

The obvious solution doesn’t work

Analogous to how we had function `B()` call function `A()` in the example above, the obvious solution would be to have one of the `Employee` constructors call the other constructor. But this will not work as intended, because constructors are not designed to be called directly from the body of another function (including other constructors)!

For example, you might think to try this:

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

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

public:
    Employee(std::string_view name)
        : m_name{ name }
    {
        std::cout << "Employee " << m_name << " created\n";
    }

    Employee(std::string_view name, int id)
        : m_name{ name }, m_id{ id }
    {
        Employee(name); // compile error
    }
};

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

This doesn’t work, and will result in a compilation error.

A more dangerous case occurs when we try to explicitly call a constructor without any arguments. This does not perform a function call to the default constructor -- rather, it creates a temporary (unnamed) object and value initializes it! Here’s a silly example that demonstrates:

```
#include <iostream>
struct Foo
{
    int x{};
    int y{};

public:
    Foo()
    {
        x = 5;
    }

    Foo(int value): y { value }
    {
        // intent: call Foo() function
        // actual: value initializes nameless temporary Foo (which is then discarded)
        Foo(); // note: this is the equivalent of writing Foo{}
    }
};

int main()
{
    Foo f{ 9 };
    std::cout << f.x << ' ' << f.y; // prints 0 9
}
```

In this example, the `Foo(int)` constructor has statement `Foo()`, expecting to call the `Foo()` constructor and assign member `x` the value `5`. However, this syntax actually creates an unnamed temporary `Foo` and then value initializes it (the same as if we had written `Foo{}`). When the `x = 5` statement executes, it is the `x` member of the temporary `Foo` that is assigned a value. Because the temporary object is not used, once it has finished construction, it is discarded.

The `x` member of the implicit object of the `Foo(int)` constructor is never assigned a value. So when we later print out its value in `main()`, we get `0` instead of the expected `5`.

Note that this case does not generate a compilation error -- rather, it just silently fails to produce the expected results!

Warning

Constructors should not be called directly from the body of another function. Doing so will either result in a compilation error, or will value initialize a temporary object and then discard it (which likely isn't what you want).

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. Applied to our example above:

```
#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 instead define a named constant and use that wherever our initialization value is needed. This allow the initialization value to be defined in one place.

The best way to do this is to use a `static constexpr` member inside the class:

```
#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 };
}
```

We cover static members in lesson 15.6 -- Static member variables.

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.

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:

```
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

Question #2

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

Show Solution