12.8 — Overlapping and delegating constructors

ALEX JULY 24, 2021

Constructors with overlapping functionality

When you instantiate a new object, the object's constructor is called implicitly. It's not uncommon to have a class with multiple constructors that have overlapping functionality. Consider the following class:

```
class Foo
{
public:
    Foo()
    {
        // code to do A
    }

    Foo(int value)
    {
        // code to do A
        // code to do B
    }
};
```

This class has two constructors: a default constructor, and a constructor that takes an integer. Because the "code to do A" portion of the constructor is required by both constructors, the code is duplicated in each constructor.

As you've (hopefully) learned by now, having duplicate code is something to be avoided as much as possible, so let's take a look at some ways to address this.

The obvious solution doesn't work

The obvious solution would be to have the Foo(int) constructor call the Foo() constructor to do the A portion.

```
class Foo
{
public:
    Foo()
    {
        // code to do A
    }

    Foo(int value)
    {
        Foo(); // use the above constructor to do A (doesn't
work)
        // code to do B
    }
};
```

However, if you try to have one constructor call another constructor in this way, it will compile and maybe cause a warning, but it will not work as you expect, and you will likely spend a long time trying to figure out why, even with a debugger. What's happening is that `Foo();` instantiates a new `Foo` object, which is immediately discarded, because it's not stored in a variable.

Delegating constructors

Constructors are allowed to call other constructors from the same class. This process is called delegating constructors (or constructor chaining).

To have one constructor call another, simply call the constructor in the member initializer list. This is one case where calling another constructor directly is acceptable. Applied to our example above:

```
class Foo
{
private:

public:
    Foo()
    {
        // code to do A
    }

    Foo(int value): Foo{} // use Foo() default constructor to
do A
    {
        // code to do B
    }

};
```

This works exactly as you'd expect. Make sure you're calling the constructor from the member initializer list, not in the body of the constructor.

Here's another example of using delegating constructors to reduce redundant code:

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

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

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

    // Use a delegating constructor to minimize redundant
code
    Employee(const std::string &name) : Employee{ 0, name }
    { }
};
```

This class has 2 constructors, one of which delegates to Employee(int, const std::string &). In this way, the amount of redundant code is minimized (we only have to write one constructor body instead of two).

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.

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 that have the same functionality, use delegating constructors to avoid duplicate code.

Using a separate function

Relatedly, you may find yourself in the situation where you want to write a member function to re-initialize a class back to default values. Because you probably already have a constructor that does this, you may be tempted to try to call the constructor from your member function. However, trying to call a constructor directly will generally result in unexpected behavior as we have shown above. Many developers simply copy the code from the constructor into the initialization function, which would work, but lead to duplicate code. The best solution in this case is to move the code from the constructor to your new function, and have the constructor call your function to do the work of "initializing" the data:

```
class Foo
{
public:
    Foo()
    {
        init();
    }

    Foo(int value)
    {
        init();
        // do something with
value
    }

    void init()
    {
        // code to "initialize"
Foo
    }
};

```

Constructors *are* allowed to call non-constructor functions in the class. Just be careful that any members the non-constructor function uses have already been initialized. Although you may be tempted to copy code from the first constructor into the second constructor, having duplicate code makes your class harder to understand and more burdensome to maintain.

We say "initialize", but it's not real initialization. By the time the constructor calls `init()`, the members already exist and have been default initialized or are uninitialized. The `init` function can only assign values to the members. There are some types that cannot be instantiated without arguments, because they don't have a default constructor. If any of the class members has such a type, the `init` function doesn't work and the constructors have to initialize those members themselves.

It is fairly common to include an init() function that initializes member variables to their default values, and then have each constructor call that init() function before doing its parameter-specific tasks. This minimizes code duplication and allows you to explicitly call init() from wherever you like.

One small caveat: be careful when using init() functions and dynamically allocated memory. Because init() functions can be called by anyone at any time, dynamically allocated memory may or may not have already been allocated when init() is called. Be careful to handle this situation appropriately -- it can be slightly confusing, since a non-null pointer could be either dynamically allocated memory or an uninitialized pointer!

Next lesson
12.9 Destructors

Back to table of contents

Previous lesson
12.7 Non-static member initialization

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ co

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

DP N N FOUT

Newest