8.6 — Overlapping and delegating constructors

BY ALEX ON SEPTEMBER 7TH, 2007 | LAST MODIFIED BY ALEX ON JULY 9TH, 2018

Constructors with overlapping functionality

When you instantiate a new object, the object's constructor is called implicitly by the C++ compiler. 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 prior to C++11

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
    }
};
```

or

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

    Foo(int value): Foo() // use the above constructor to do A (doesn't work prior to C++11)
    {
        // code to do B
    }
};
```

However, with a pre-C++11 compiler, if you try to have one constructor call another constructor, it will often compile, 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.

(Optional explanation: Prior to C++11, calling a constructor explicitly from another constructor creates a temporary object, initializes the temporary object using the constructor, and then discards it, leaving your original object unchanged)

Using a separate function

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. The best solution to this issue is to create a non-constructor function that does the common initialization, and have both constructors call that function.

Given this, we can change the above class to the following:

```
class Foo
{
private:
    void DoA()
    {
        // code to do A
    }

public:
    Foo()
    {
        DoA();
    }

    Foo(int nValue)
    {
        DoA();
        // code to do B
    }

};
```

In this way, code duplication is kept to a minimum.

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. Many developers simply copy the code from the constructor in your 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 init Foo
    }
};
```

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!

Delegating constructors in C++11

Starting with C++11, constructors are now allowed to call other constructors. 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 constructor to reduce redundant code:

```
#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 constructors 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.


8.7 -- Destructors


Index


8.5b -- Non-static member initialization

Share this:

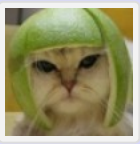
Facebook Twitter Google Pinterest

C++ TUTORIAL | PRINT THIS POST

75 comments to 8.6 — Overlapping and delegating constructors

hrmn
July 6, 2018 at 9:37 am · Reply

in code above there's a called constructor in the body of constructor, after 'or' ,
then below while explaining delegating it's
Make sure you're calling the constructor from the member initializer list, not in the body of the constructor.
am i missing something , am confused!
we can call constructor in body of constructor after c++11 right?
Thanks



Alex
July 9, 2018 at 1:49 pm · Reply

No, you can't. The example you're referencing is showing you what you shouldn't do. I've updated the lesson to try to make that slightly more clear.

Jack
June 25, 2018 at 7:24 am · Reply

Noticed that in the example:

```
1  class Foo
2  {
3  public:
4      Foo()
5      {
6          Init();
7      }
8
9      Foo(int value)
10     {
11         Init();
12         // do something with value
```