8.6 — Overlapping and delegating constructors

BY ALEX ON SEPTEMBER 7TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

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;
```

```
9
10  public:
11      Employee(int id=0, const std::string &name=""):
12          m_id(id), m_name(name)
13      {
14          std::cout << "Employee " << m_name << " created.\n";
15      }
16
17      // Use a delegating constructors to minimize redundant code
18      Employee(const std::string &name) : Employee(0, name) { }
19  };
```

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**

C++ TUTORIAL | PRINT THIS POST

106 comments to 8.6 — Overlapping and delegating constructors

« Older Comments 1 2

Samira Ferdi
December 17, 2019 at 5:22 pm · Reply

Hi, Alex and Nascardriver!

Can we call the method using constructor through initialize list?

```
void method();

Constructor(): method()
{
//or should I call in the constructor's body
}
```

nascardriver
December 18, 2019 at 4:34 am · Reply

No, you have to call it in the body.

Nguyen
August 28, 2019 at 3:55 pm · Reply

Hi,

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

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

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

    Employee(const std::string &name = "Alex") : Employee(0, name) { }
};
int main()
{
    Employee myName();
    return 0;
}
```

I expected to see my output shown as following:
Employee Alex created.

But there is no output at all? Please help.

nascardriver
August 29, 2019 at 12:08 am · Reply

Line 21 is a function prototype. Use brace initialization.

Ganesh Koli
August 6, 2019 at 6:38 am · Reply

Hi Alex, Thank you for the article.
How this will work in inheritance ? and want to initialize super class members as well?

Class Derive : public B

It is not allowing the other member initialization with delegate constructor.
I think this restriction to avoid below situation :

```
Class Derive : public Base
{
  int size;
public :
  Derive () : Base () {}
  Derive (int s) : Base(s), Derive() {}   // compilation error
}
```

In above code if compilation allow it then Base class constructor could called two times ? (or something danger)

Please correct me , if i am wrong or understood it wrongly .

nascardriver
August 6, 2019 at 6:41 am · Reply

`Derive` can call `B`'s constructors, but it can't initialize `B`'s members in the member initializer list.

Jeffrey Liu
July 30, 2019 at 8:34 am · Reply

Hi,

When using a non-constructor function to avoid duplicate code, would this not lead to certain issues (the same as the ones given when not using a member initializer list) because they are assigning values to the member variables, and not actually initializing them?

Thank you!

Alex
August 3, 2019 at 8:45 pm · Reply

Yep. That may or may not be a problem, depending on the specific members of the class.

Dany
July 10, 2019 at 3:55 am · Reply

Hello. Can someone clarify which syntax should i prefer upon creation of object (i think {} because we can distinguish initialization from assignment and function call)
and upon creation of delegating constructor (here i am really confused why both versions are available). Thanks in advance.

```
1 #include <string>
2 #include <iostream>
```

```
3
4   class Employee
5   {
6   private:
7       int m_id;
8       std::string m_name;
9
10  public:
11      Employee(int id = 0, const std::string & name = "") :
12          m_id(id), m_name(name)
13      {
14          std::cout << "Employee " << m_name << " created.\n";
15      }
16
17      // Use a delegating constructors to minimize redundant code
18      //which one i should prefer?
19      Employee(const std::string& name) : Employee{ 0, name } { }
20      Employee(const std::string& name) : Employee( 0, name ) { }
21  };
22
23  int main()
24  {
25      //which one i should prefer?
26      Employee emp1("George");
27      Employee emp2{ "Jack" };
28  }
```

nascardriver
July 10, 2019 at 5:06 am · Reply

It's not just the syntax that's different, the 2 forms can do different things.
Both are direct initialization, but brace initialization (curly braces) enforces stricter type checks and can be used to initialize lists or default-initialize.
Use brace initialization unless you're constructing a type which has a list constructor and you want to use a non-list constructor.

```
1   #include <string>
2   #include <iostream>
3
4   class Employee
5   {
6   private:
7       int m_id;
8       std::string m_name;
9
10  public:
11      Employee(int id = 0, const std::string & name = "") :
12          m_id{ id }, m_name{ name } // Those too
13      {
14          std::cout << "Employee " << m_name << " created.\n";
15      }
16
17      // Use a delegating constructors to minimize redundant code
18      Employee(const std::string& name) : Employee{ 0, name } { }
19      // Employee(const std::string& name) : Employee( 0, name ) { }
20  };
21
22  int main()
23  {
24      // Employee emp1("George");
25      Employee emp2{ "Jack" };
```

```
26 }
```

Yaroslav
July 3, 2019 at 2:35 pm · Reply

(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)

May be it will be better to rewrite this because for me (for a noob) it was not clear at all what's going on.

may be something like:
prior to c++11 there was no member initializing list so we can't use it then.
and if we will try to call the first constructor from a body of another constructor it will not call it, it will just simply create another instance of a class (in a form of anonymous object, you will learn it in ...), then it will initialize that copy/instance with the first constructor, and then discard it, leaving our original object unchanged.

Alireza
June 20, 2019 at 3:57 am · Reply

Hi,
Why can't I do this way:

```
1  class Ball
2  {
3  private:
4      std::string m_color{"black"};
5      double m_radius{10.0};
6  public:
7      Ball()
8      {}
9      Ball(const std::string &color)
10     {
11         m_color = color;
12     }
13     Ball(double radius)
14     {
15         m_radius = radius;
16     }
17     Ball(const std::string &color, double radius)
18         :Ball{color}, Ball{radius}
19     { }
20
21     void print()
22     {
23         std::cout << "color: " << m_color << ", radius: " << m_radius << '\n';
24     }
25 };
```

Causes an error:

```
1  main.cpp:18: error: an initializer for a delegating constructor must appear alone
```

What does it mean ?

nascardriver
June 20, 2019 at 4:10 am · Reply

If you delegate construction to another constructor, that constructor call must be the only thing in the constructor initializer list. There can be no other initializations or constructor calls.
Once you call a constructor, the object is considered initialized. Since an object can only be initialized once, you can't initialize anything else (eg. by calling more constructors).
Initialize `m_color` and `m_radius` directly.

Alireza
June 20, 2019 at 7:17 am · Reply

So a constructor can't be called more than once in initializer list in other constructor, right ??
Thanks a lot

nascardriver
June 20, 2019 at 7:21 am · Reply

right

Tommy
April 28, 2019 at 3:18 am · Reply

I'm failing to understand your employee example. The delegating constructor (ie. the extension, right?), it looks redundant because it's not adding any new functionality. It seems like it has less functionality actually. Have I missed something? How is the first constructor not enough?

Alex
May 1, 2019 at 2:57 pm · Reply

The first constructor won't let you create Employee("Alex").
It's not a wonderful example because this could be solved trivially by flipping the order of the id and string fields, but it does at least show the mechanics.

NXPY
April 15, 2019 at 10:26 pm · Reply

When using a seperate function , isn't it better to make it inline to reduce time ?

« Older Comments 1 2