# 11.4 — Constructors and initialization of derived classes

BY ALEX ON JANUARY 9TH, 2008 | LAST MODIFIED BY NASCARDRIVER ON JANUARY 27TH, 2020

In the past two lessons, we've explored some basics around inheritance in C++ and the order that derived classes are initialized. In this lesson, we'll take a closer look at the role of constructors in the initialization of derived classes. To do so, we will continue to use the simple Base and Derived classes we developed in the previous lesson:

```
class Base
{
public:
    int m_id;

    Base(int id=0)
        : m_id{ id }
    {
    }

    int getId() const { return m_id; }
};

class Derived: public Base
{
public:
    double m_cost;

    Derived(double cost=0.0)
        : m_cost{ cost }
    {
    }

    double getCost() const { return m_cost; }
};
```

With non-derived classes, constructors only have to worry about their own members. For example, consider Base. We can create a Base object like this:

```
int main()
{
    Base base{ 5 }; // use Base(int) constructor

    return 0;
}
```

Here's what actually happens when base is instantiated:

1. Memory for base is set aside
2. The appropriate Base constructor is called
3. The initialization list initializes variables
4. The body of the constructor executes
5. Control is returned to the caller

This is pretty straightforward. With derived classes, things are slightly more complex:

```
int main()
{
    Derived derived{ 1.3 }; // use Derived(double) constructor

    return 0;
```

```
}
```

Here's what actually happens when derived is instantiated:

1. Memory for derived is set aside (enough for both the Base and Derived portions)
2. The appropriate Derived constructor is called
3. **The Base object is constructed first using the appropriate Base constructor**. If no base constructor is specified, the default constructor will be used.
4. The initialization list initializes variables
5. The body of the constructor executes
6. Control is returned to the caller

The only real difference between this case and the non-inherited case is that before the Derived constructor can do anything substantial, the Base constructor is called first. The Base constructor sets up the Base portion of the object, control is returned to the Derived constructor, and the Derived constructor is allowed to finish up its job.

**Initializing base class members**

One of the current shortcomings of our Derived class as written is that there is no way to initialize m_id when we create a Derived object. What if we want to set both m_cost (from the Derived portion of the object) and m_id (from the Base portion of the object) when we create a Derived object?

New programmers often attempt to solve this problem as follows:

```
class Derived: public Base
{
public:
    double m_cost;

    Derived(double cost=0.0, int id=0)
        // does not work
        : m_cost{ cost }, m_id{ id }
    {
    }

    double getCost() const { return m_cost; }
};
```

This is a good attempt, and is almost the right idea. We definitely need to add another parameter to our constructor, otherwise C++ will have no way of knowing what value we want to initialize m_id to.

However, C++ prevents classes from initializing inherited member variables in the initialization list of a constructor. In other words, the value of a variable can only be set in an initialization list of a constructor belonging to the same class as the variable.

Why does C++ do this? The answer has to do with const and reference variables. Consider what would happen if m_id were const. Because const variables must be initialized with a value at the time of creation, the base class constructor must set its value when the variable is created. However, when the base class constructor finishes, the derived class constructors initialization lists are then executed. Each derived class would then have the opportunity to initialize that variable, potentially changing its value! By restricting the initialization of variables to the constructor of the class those variables belong to, C++ ensures that all variables are initialized only once.

The end result is that the above example does not work because m_id was inherited from Base, and only non-inherited variables can be initialized in the initialization list.

However, inherited variables can still have their values changed in the body of the constructor using an assignment. Consequently, new programmers often also try this:

```
class Derived: public Base
{
public:
```

```
    double m_cost;

    Derived(double cost=0.0, int id=0)
        : m_cost{ cost }
    {
        m_id = id;
    }

    double getCost() const { return m_cost; }
};
```

While this actually works in this case, it wouldn't work if m_id were a const or a reference (because const values and references have to be initialized in the initialization list of the constructor). It's also inefficient because m_id gets assigned a value twice: once in the initialization list of the Base class constructor, and then again in the body of the Derived class constructor. And finally, what if the Base class needed access to this value during construction? It has no way to access it, since it's not set until the Derived constructor is executed (which pretty much happens last).

So how do we properly initialize m_id when creating a Derived class object?

In all of the examples so far, when we instantiate a Derived class object, the Base class portion has been created using the default Base constructor. Why does it always use the default Base constructor? Because we never told it to do otherwise!

Fortunately, C++ gives us the ability to explicitly choose which Base class constructor will be called! To do this, simply add a call to the base class Constructor in the initialization list of the derived class:

```
class Derived: public Base
{
public:
    double m_cost;

    Derived(double cost=0.0, int id=0)
        : Base{ id }, // Call Base(int) constructor with value id!
          m_cost{ cost }
    {
    }

    double getCost() const { return m_cost; }
};
```

Now, when we execute this code:

```
int main()
{
    Derived derived{ 1.3, 5 }; // use Derived(double, int) constructor
    std::cout << "Id: " << derived.getId() << '\n';
    std::cout << "Cost: " << derived.getCost() << '\n';

    return 0;
}
```

The base class constructor Base(int) will be used to initialize m_id to 5, and the derived class constructor will be used to initialize m_cost to 1.3!

Thus, the program will print:

```
Id: 5
Cost: 1.3
```

In more detail, here's what happens:

1. Memory for derived is allocated.
2. The Derived(double, int) constructor is called, where cost = 1.3, and id = 5
3. The compiler looks to see if we've asked for a particular Base class constructor. We have! So it calls Base(int) with id = 5.
4. The base class constructor initialization list sets m_id to 5
5. The base class constructor body executes, which does nothing
6. The base class constructor returns
7. The derived class constructor initialization list sets m_cost to 1.3
8. The derived class constructor body executes, which does nothing
9. The derived class constructor returns

This may seem somewhat complex, but it's actually very simple. All that's happening is that the Derived constructor is calling a specific Base constructor to initialize the Base portion of the object. Because m_id lives in the Base portion of the object, the Base constructor is the only constructor that can initialize that value.

Note that it doesn't matter where in the Derived constructor initialization list the Base constructor is called -- it will always execute first.

**Now we can make our members private**

Now that you know how to initialize base class members, there's no need to keep our member variables public. We make our member variables private again, as they should be.

As a quick refresher, public members can be accessed by anybody. Private members can only be accessed by member functions of the same class. Note that this means derived classes can not access private members of the base class directly! Derived classes will need to use access functions to access private members of the base class.

Consider:

```
#include <iostream>

class Base
{
private: // our member is now private
    int m_id;

public:
    Base(int id=0)
        : m_id{ id }
    {
    }

    int getId() const { return m_id; }
};

class Derived: public Base
{
private: // our member is now private
    double m_cost;

public:
    Derived(double cost=0.0, int id=0)
        : Base{ id }, // Call Base(int) constructor with value id!
            m_cost{ cost }
    {
    }

    double getCost() const { return m_cost; }
};
```

```
31
32  int main()
33  {
34      Derived derived{ 1.3, 5 }; // use Derived(double, int) constructor
35      std::cout << "Id: " << derived.getId() << '\n';
36      std::cout << "Cost: " << derived.getCost() << '\n';
37
38      return 0;
39  }
```

In the above code, we've made m_id and m_cost private. This is fine, since we use the relevant constructors to initialize them, and use a public accessor to get the values.

This prints, as expected:

```
Id: 5
Cost: 1.3
```

We'll talk more about access specifiers in the next lesson.

**Another example**

Let's take a look at another pair of classes we've previously worked with:

```
1   #include <string>
2
3   class Person
4   {
5   public:
6       std::string m_name;
7       int m_age;
8
9       Person(const std::string& name = "", int age = 0)
10          : m_name{ name }, m_age{ age }
11      {
12      }
13
14      const std::string& getName() const { return m_name; }
15      int getAge() const { return m_age; }
16  };
17
18  // BaseballPlayer publicly inheriting Person
19  class BaseballPlayer : public Person
20  {
21  public:
22      double m_battingAverage;
23      int m_homeRuns;
24
25      BaseballPlayer(double battingAverage = 0.0, int homeRuns = 0)
26          : m_battingAverage{ battingAverage },
27            m_homeRuns{ homeRuns }
28      {
29      }
30  };
```

As we'd previously written it, BaseballPlayer only initializes its own members and does not specify a Person constructor to use. This means every BaseballPlayer we create is going to use the default Person constructor, which will initialize the name to blank and age to 0. Because it makes sense to give our BaseballPlayer a name and age when we create them, we should modify this constructor to add those parameters.

Here's our updated classes that use private members, with the BaseballPlayer class calling the appropriate Person constructor to initialize the inherited Person member variables:

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

class Person
{
private:
    std::string m_name;
    int m_age;

public:
    Person(const std::string& name = "", int age = 0)
        : m_name{ name }, m_age{ age }
    {
    }

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

};
// BaseballPlayer publicly inheriting Person
class BaseballPlayer : public Person
{
private:
    double m_battingAverage;
    int m_homeRuns;

public:
    BaseballPlayer(const std::string& name = "", int age = 0,
        double battingAverage = 0.0, int homeRuns = 0)
        : Person{ name, age }, // call Person(const std::string&, int) to initialize these fi
elds
            m_battingAverage{ battingAverage }, m_homeRuns{ homeRuns }
    {
    }

    double getBattingAverage() const { return m_battingAverage; }
    int getHomeRuns() const { return m_homeRuns; }
};
```

Now we can create baseball players like this:

```
int main()
{
    BaseballPlayer pedro{ "Pedro Cerrano", 32, 0.342, 42 };

    std::cout << pedro.getName() << '\n';
    std::cout << pedro.getAge() << '\n';
    std::cout << pedro.getHomeRuns() << '\n';

    return 0;
}
```

This outputs:

```
Pedro Cerrano
32
42
```

As you can see, the name and age from the base class were properly initialized, as was the number of home runs from the derived class.

**Inheritance chains**

Classes in an inheritance chain work in exactly the same way.

```
#include <iostream>

class A
{
public:
    A(int a)
    {
        std::cout << "A: " << a << '\n';
    }
};

class B: public A
{
public:
    B(int a, double b)
    : A{ a }
    {
        std::cout << "B: " << b << '\n';
    }
};

class C: public B
{
public:
    C(int a , double b , char c)
    : B{ a, b }
    {
        std::cout << "C: " << c << '\n';
    }
};

int main()
{
    C c{ 5, 4.3, 'R' };

    return 0;
}
```

In this example, class C is derived from class B, which is derived from class A. So what happens when we instantiate an object of class C?

First, main() calls C(int, double, char). The C constructor calls B(int, double). The B constructor calls A(int). Because A does not inherit from anybody, this is the first class we'll construct. A is constructed, prints the value 5, and returns control to B. B is constructed, prints the value 4.3, and returns control to C. C is constructed, prints the value 'R', and returns control to main(). And we're done!

Thus, this program prints:

```
A: 5
B: 4.3
C: R
```

It is worth mentioning that constructors can only call constructors from their immediate parent/base class. Consequently, the C constructor could not call or pass parameters to the A constructor directly. The C constructor can only call the B constructor (which has the responsibility of calling the A constructor).

**Destructors**

When a derived class is destroyed, each destructor is called in the *reverse* order of construction. In the above example, when c is destroyed, the C destructor is called first, then the B destructor, then the A destructor.

**Summary**

When constructing a derived class, the derived class constructor is responsible for determining which base class constructor is called. If no base class constructor is specified, the default base class constructor will be used. In that case, if no default base class constructor can be found (or created by default), the compiler will display an error. The classes are then constructed in order from most base to most derived.

At this point, you now understand enough about C++ inheritance to create your own inherited classes!

**Quiz time!**

1) Let's implement our Fruit example that we talked about in our introduction to inheritance. Create a Fruit base class that contains two private members: a name (std::string), and a color (std::string). Create an Apple class that inherits Fruit. Apple should have an additional private member: fiber (double). Create a Banana class that also inherits Fruit. Banana has no additional members.

The following program should run:

```
int main()
{
    const Apple a{ "Red delicious", "red", 4.2 };
    std::cout << a << '\n';

    const Banana b{ "Cavendish", "yellow" };
    std::cout << b << '\n';

    return 0;
}
```

And print the following:

```
Apple(Red delicious, red, 4.2)
Banana(Cavendish, yellow)
```

Hint: Because a and b are const, you'll need to mind your const's. Make sure your parameters and functions are appropriately const.

**Show Solution**

**11.5 -- Inheritance and access specifiers**

**Index**