

11.4 — Constructors and initialization of derived classes

BY ALEX ON JANUARY 9TH, 2008 | LAST MODIFIED BY ALEX ON DECEMBER 28TH, 2015

In the past two lessons, we've looked at some basics about inheritance in C++ and explored 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 class we developed in the previous lesson:

```

1  class Base
2  {
3  public:
4      int m_nValue;
5
6      Base(int nValue=0)
7          : m_nValue(nValue)
8      {
9      }
10 };
11
12 class Derived: public Base
13 {
14 public:
15     double m_dValue;
16
17     Derived(double dValue=0.0)
18         : m_dValue(dValue)
19     {
20     }
21 };

```

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:

```

1  int main()
2  {
3      Base cBase(5); // use Base(int) constructor
4
5      return 0;
6  }

```

Here's what actually happens when cBase is instantiated:

1. Memory for cBase 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:

```

1  int main()
2  {
3      Derived cDerived(1.3); // use Derived(double) constructor
4
5      return 0;
6  }

```

Here's what actually happens when cDerived is instantiated:

1. Memory for cDerived 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**

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 it's 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_nValue` when we create a Derived object. What if we want to set both `m_dValue` (from the Derived portion of the object) and `m_nValue` (from the Base portion of the object) when we create a Derived object?

New programmers often attempt to solve this problem as follows:

```

1  class Derived: public Base
2  {
3  public:
4      double m_dValue;
5
6      Derived(double dValue=0.0, int nValue=0)
7          // does not work
8          : m_dValue(dValue), m_nValue(nValue)
9      {
10     }
11 };

```

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_nValue` 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_nValue` 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 it's 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_nValue` was inherited from Base, and only non-inherited variables can be changed 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:

```

1  class Derived: public Base
2  {
3  public:
4      double m_dValue;
5
6      Derived(double dValue=0.0, int nValue=0)
7          : m_dValue(dValue)
8      {
9          m_nValue = nValue;
10     }
11 };

```

While this actually works in this case, it wouldn't work if `m_nValue` 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_nValue` 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.

So how do we properly initialize `m_nValue` 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:

```

1  class Derived: public Base
2  {
3  public:
4      double m_dValue;
5
6      Derived(double dValue=0.0, int nValue=0)
7          : Base(nValue), // Call Base(int) constructor with value nValue!
8            m_dValue(dValue)
9      {
10     }
11 };

```

Now, when we execute this code:

```

1  int main()
2  {
3      Derived cDerived(1.3, 5); // use Derived(double) constructor
4
5      return 0;
6  }

```

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

In more detail, here's what happens:

1. Memory for `cDerived` is allocated.
2. The `Derived(double, int)` constructor is called, where `dValue = 1.3`, and `nValue = 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 `nValue = 5`.
4. The base class constructor initialization list sets `m_nValue` to 5
5. The base class constructor body executes
6. The base class constructor returns
7. The derived class constructor initialization list sets `m_dValue` to 1.3
8. The derived class constructor body executes
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_nValue` lives in the Base portion of the object, the Base constructor is the only constructor that can initialize it's value.

Another example

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

```

1  #include <string>
2  class Person
3  {
4  public:
5      std::string m_strName;
6      int m_nAge;
7      bool m_bIsMale;
8
9      std::string GetName() { return m_strName; }

```

```

10     int GetAge() { return m_nAge; }
11     bool IsMale() { return m_bIsMale; }
12
13     Person(std::string strName = "", int nAge = 0, bool bIsMale = false)
14         : m_strName(strName), m_nAge(nAge), m_bIsMale(bIsMale)
15     {
16     }
17 };
18
19 // BaseballPlayer publicly inheriting Person
20 class BaseballPlayer : public Person
21 {
22 public:
23     double m_dBattingAverage;
24     int m_nHomeRuns;
25
26     BaseballPlayer(double dBattingAverage = 0.0, int nHomeRuns = 0)
27         : m_dBattingAverage(dBattingAverage), m_nHomeRuns(nHomeRuns)
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 new `BaseballPlayer` class with a constructor that calls the `Person` constructor to initialize the inherited `Person` member variables.

```

1 // BaseballPlayer publicly inheriting Person
2 class BaseballPlayer : public Person
3 {
4 public:
5     double m_dBattingAverage;
6     int m_nHomeRuns;
7
8     BaseballPlayer(std::string strName = "", int nAge = 0, bool bIsMale = false,
9                     double dBattingAverage = 0.0, int nHomeRuns = 0)
10        : Person(strName, nAge, bIsMale), // call Person(std::string, int, bool) to initialize
11        these fields
12        m_dBattingAverage(dBattingAverage), m_nHomeRuns(nHomeRuns)
13    {
14    }
15 };

```

Now we can create baseball players like this:

```

1 int main()
2 {
3     BaseballPlayer cPlayer("Pedro Cerrano", 32, true, 0.342, 42);
4
5     return 0;
6 }

```

To prove that it works:

```

1 int main()
2 {
3     BaseballPlayer cPlayer("Pedro Cerrano", 32, true, 0.342, 42);
4
5     using namespace std;
6     cout << cPlayer.m_strName << endl;
7     cout << cPlayer.m_nAge << endl;
8     cout << cPlayer.m_nHomeRuns;
9 }

```

```
10     return 0;
11 }
```

This outputs:

```
Pedro Cerrano
32
42
```

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

Inheritance chains

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

```
1  #include <iostream>
2  using namespace std;
3
4  class A
5  {
6  public:
7      A(int nValue)
8      {
9          cout << "A: " << nValue << endl;
10     }
11 };
12
13 class B: public A
14 {
15 public:
16     B(int nValue, double dValue)
17     : A(nValue)
18     {
19         cout << "B: " << dValue << endl;
20     }
21 };
22
23 class C: public B
24 {
25 public:
26     C(int nValue, double dValue, char chValue)
27     : B(nValue, dValue)
28     {
29         cout << "C: " << chValue << endl;
30     }
31 };
32
33 int main()
34 {
35     C cClass(5, 4.3, 'R');
36
37     return 0;
38 }
```

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 is not inherited, 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 cClass is destroyed, the C destructor is called first, then the B destructor, then the A destructor.

Summary

Although it is true that the most base class is initialized first, this actually only happens after each constructor has called the parent constructor in turn. This gives us the opportunity to specify which of the parent's constructors we want to use to initialize inherited members. Once the base constructor has finished constructing the base portion of the class, control returns to the derived constructor and it executes as normal.

One of the primary advantages of using a base class constructor to initialize the base class members is that if the base class constructor is ever changed, both the base class and all inherited classes will automatically use the changes! This helps keep maintenance and duplicate code down.



11.5 -- Inheritance and access specifiers




Index



11.3 -- Order of construction of derived classes

Share this:

 Email

 Facebook 8

 Twitter

 Google

 Pinterest