

24.4 — Constructors and initialization of derived classes

👤 ALEX¹ 🕒 SEPTEMBER 11, 2023

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:

```
1 class Base
2 {
3 public:
4     int m_id {};;
5
6     Base(int id=0)
7         : m_id{ id }
8     {
9     }
10
11     int getId() const { return m_id; }
12 };
13
14 class Derived: public Base
15 {
16 public:
17     double m_cost {};;
18
19     Derived(double cost=0.0)
20         : m_cost{ cost }
21     {
22     }
23
24     double getCost() const { return m_cost; }
25 };
```

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 base{ 5 }; // use Base(int) constructor
4
5     return 0;
6 }
```

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 member initializer 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 derived{ 1.3 }; // use Derived(double) constructor
4 |
5 |     return 0;
6 | }
```

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 member initializer 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:

```
1 | class Derived: public Base
2 | {
3 | public:
4 |     double m_cost {};;
5 |
6 |     Derived(double cost=0.0, int id=0)
7 |         // does not work
8 |         : m_cost{ cost }
9 |         , m_id{ id }
10 |    {
11 |    }
12 |
13 |     double getCost() const { return m_cost; }
14 | };
```

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 member initializer list of a constructor. In other words, the value of a member variable can only be set in a member initializer 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 constructor's member initializer 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 member initializer 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_cost {};
5
6     Derived(double cost=0.0, int id=0)
7         : m_cost{ cost }
8     {
9         m_id = id;
10    }
11
12    double getCost() const { return m_cost; }
13 };
```

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 member initializer list of the constructor). It's also inefficient because `m_id` gets assigned a value twice: once in the member initializer 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 member initializer list of the derived class:

```
1 class Derived: public Base
2 {
3 public:
4     double m_cost {};
5
6     Derived(double cost=0.0, int id=0)
7         : Base{ id } // Call Base(int) constructor with value id!
8         , m_cost{ cost }
9     {
10    }
11
12    double getCost() const { return m_cost; }
13 };
```

Now, when we execute this code:

```
1 | #include <iostream>
2 |
3 | int main()
4 | {
5 |     Derived derived{ 1.3, 5 }; // use Derived(double, int) constructor
6 |     std::cout << "Id: " << derived.getId() << '\n';
7 |     std::cout << "Cost: " << derived.getCost() << '\n';
8 |
9 |     return 0;
10 | }
```

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 member initializer 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 member initializer 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 member initializer 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:

```

1  #include <iostream>
2
3  class Base
4  {
5  private: // our member is now private
6      int m_id {};
7
8  public:
9      Base(int id=0)
10         : m_id{ id }
11     {
12     }
13
14     int getId() const { return m_id; }
15 };
16
17 class Derived: public Base
18 {
19 private: // our member is now private
20     double m_cost;
21
22 public:
23     Derived(double cost=0.0, int id=0)
24         : Base{ id } // Call Base(int) constructor with value id!
25         , m_cost{ cost }
26     {
27     }
28
29     double getCost() const { return m_cost; }
30 };
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 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  #include <string_view>
3
4  class Person
5  {
6  public:
7      std::string m_name;
8      int m_age {};
9
10     Person(std::string_view name = "", int age = 0)
11         : m_name{ name }, m_age{ age }
12     {
13     }
14
15     const std::string& getName() const { return m_name; }
16     int getAge() const { return m_age; }
17 };
18
19 // BaseballPlayer publicly inheriting Person
20 class BaseballPlayer : public Person
21 {
22 public:
23     double m_battingAverage {};
24     int m_homeRuns {};
25
26     BaseballPlayer(double battingAverage = 0.0, int homeRuns = 0)
27         : m_battingAverage{ battingAverage },
28           m_homeRuns{ homeRuns }
29     {
30     }
31 };

```

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:

```

1  #include <iostream>
2  #include <string>
3  #include <string_view>
4
5  class Person
6  {
7  private:
8      std::string m_name;
9      int m_age {};
10
11 public:
12     Person(std::string_view name = "", int age = 0)
13         : m_name{ name }, m_age{ age }
14     {
15     }
16
17     const std::string& getName() const { return m_name; }
18     int getAge() const { return m_age; }
19
20 };
21 // BaseballPlayer publicly inheriting Person
22 class BaseballPlayer : public Person
23 {
24 private:
25     double m_battingAverage {};
26     int m_homeRuns {};
27
28 public:
29     BaseballPlayer(std::string_view name = "", int age = 0,
30         double battingAverage = 0.0, int homeRuns = 0)
31         : Person{ name, age } // call Person(std::string_view, int) to initialize
32         these fields
33         , m_battingAverage{ battingAverage }, m_homeRuns{ homeRuns }
34     {
35     }
36
37     double getBattingAverage() const { return m_battingAverage; }
38     int getHomeRuns() const { return m_homeRuns; }
39 };

```

Now we can create baseball players like this:

```

1  #include <iostream>
2
3  int main()
4  {
5      BaseballPlayer pedro{ "Pedro Cerrano", 32, 0.342, 42 };
6
7      std::cout << pedro.getName() << '\n';
8      std::cout << pedro.getAge() << '\n';
9      std::cout << pedro.getBattingAverage() << '\n';
10     std::cout << pedro.getHomeRuns() << '\n';
11
12     return 0;
13 }

```

This outputs:

```

Pedro Cerrano
32
0.342
42

```

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

Inheritance chains

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

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

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.

Warning

If your base class has virtual functions, your destructor should also be virtual, otherwise undefined behavior will result in certain cases. We cover this case in lesson [25.4 -- Virtual destructors, virtual assignment, and overriding virtualization](https://www.learncpp.com/cpp-tutorial/virtual-destructors-virtual-assignment-and-overriding-virtualization/) (<https://www.learncpp.com/cpp-tutorial/virtual-destructors-virtual-assignment-and-overriding-virtualization/>)².

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:

```
1 | #include <iostream>
2 |
3 | int main()
4 | {
5 |     const Apple a{ "Red delicious", "red", 4.2 };
6 |     std::cout << a << '\n';
7 |
8 |     const Banana b{ "Cavendish", "yellow" };
9 |     std::cout << b << '\n';
10 |
11 |     return 0;
12 | }
```

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 consts. Make sure your parameters and functions are appropriately const.

[Show Solution](#) (javascript:void(0))³



[Next lesson](#)

24.5 [Inheritance and access specifiers](#)

4



[Back to table of contents](#)

5



[Previous lesson](#)

24.3 [Order of construction of derived classes](#)

6

7






- B
- U
- URL
- INLINE CODE
- C++ CODE BLOCK
- HELP!


Leave a comment...

 Name*

@ Email* 

 Find a mistake? Leave a comment above!

 Avatars from <https://gravatar.com/>⁹ are connected to your provided email address.

Notify me about replies: 

POST COMMENT

279 COMMENTS

Newest ▼



Nidhi Gupta
🕒 May 5, 2025 1:30 pm PDT

here are the points i would say i get:

Base class construction always happens first, regardless of the order of initializer list entries.

Inherited variables must be initialized by their own class constructor, not by the derived class. This ensures proper initialization of const or reference members.

Derived constructors can choose which base constructor to call by explicitly invoking it in their initializer list.

Private members of the base class cannot be accessed directly by derived classes—accessor methods must be used.

Destructors are called in the reverse order of construction, ensuring proper resource cleanup.

 1  Reply



Aiden

 March 22, 2025 12:22 am PDT


```

1  #include <iostream>
2  #include <string>
3  #include <string_view>
4
5  class Fruit
6  {
7  private:
8      const std::string m_name{};
9      const std::string m_color{};
10
11 public:
12     Fruit(std::string_view name, std::string_view color)
13         : m_name(name), m_color(color)
14     {
15
16     }
17
18     const std::string_view getName() const { return m_name; }
19     const std::string_view getColor() const { return m_color; }
20 };
21
22 class Apple : public Fruit
23 {
24 private:
25     const double fiber{};
26 public:
27     Apple(std::string_view name, std::string_view color, double fiber)
28         : Fruit(name, color), fiber(fiber)
29     {
30     }
31
32     const double getFiber() const { return fiber; }
33 };
34
35 class Banana : public Fruit
36 {
37 public:
38     Banana(std::string_view name, std::string_view color)
39         : Fruit(name, color)
40     {
41     }
42 };
43
44 std::ostream& operator<<(std::ostream& out, const Apple& apple)
45 {
46     out << "Apple (" << apple.getName() << ", " << apple.getColor()
47         << ", " << apple.getFiber() << ")";
48
49     return out;
50 }
51
52 std::ostream& operator<<(std::ostream& out, const Banana& banana)
53 {
54     out << "Banana (" << banana.getName() << ", " << banana.getColor()
55         << ")";
56
57     return out;
58 }
59
60
61 int main()
62 {
63     const Apple a{ "Red delicious", "red", 4.2 };
64     std::cout << a << '\n';
65
66     const Banana b{ "Cavendish", "yellow" };
67     std::cout << b << '\n';
68
69     return 0;
70 }

```

1 Reply



Nyac

🕒 February 26, 2025 12:55 am PST

```
1  //This this anwer of quiz time
2
3  #include <iostream>
4  #include <ostream>
5  #include <string_view>
6
7  class fruit
8  {
9      private:
10         std::string furuit_name{};
11         std::string furit_colour{};
12     public:
13         fruit(std::string_view name="",std::string_view colour="")
14             :furuit_name(name),furit_colour(colour){}
15         const auto& fruitName()const{return furuit_name;}
16         const auto& fruotColur()const{return furit_colour;}
17     };
18     class apple:public fruit
19     {
20     private:
21         double m_fiber{};
22     public:
23         apple(std::string_view name="",std::string colour="",double fiber=0)
24             :fruit(name,colur),m_fiber(fiber){}
25         auto operator()()const{return m_fiber;}
26         friend std::ostream&operator<<(std::ostream&out,const apple&a)
27         {
28             out<<"Apple("<<a.fruitName()<<','<<a.fruotColur()<<','<<a()<<')';
29             return out;
30         }
31     };
32     class banana:public fruit
33     {
34     public:
35         banana(std::string_view names="",std::string_view colour="")
36             :fruit(names,colour)
37         {}
38         friend std::ostream&operator<<(std::ostream&out,const banana&b)
39         {
40             out<<"Banna("<<b.fruitName()<<','<<b.fruotColur()<<')';
41             return out;
42         }
43     };
44
45     int main()
46     {
47         const apple a{"Red delicious","red",5.6};
48         std::cout<<a<<'\n';
49         const banana b{"cavendish","yellow"};
50         std::cout<<b<<'\n';
51     }
52 }
```

👍 0 ➡ Reply



Asicx

🕒 September 16, 2024 4:52 pm PDT

I tried this so i don't have to write two "<<" overloads but ended with an "ambiguous" error:

```
1  template <typename T>
2  std::ostream& operator<<(std::ostream& out, const T& myClass)
3  {
4
5      if constexpr (std::is_same_v<T, Apple>)
6      {
7          out << "Apple(" << myClass.getName() << ", " << myClass.getColor() << ", " <<
myClass.getFibersCount() << ')';
8      }
9      else if constexpr (std::is_same_v<T, Banana>)
10     {
11         out << "Banana(" << myClass.getName() << ", " << myClass.getColor() << ')';
12     }
13
14     return out;
15 }
```

Is there any solution to make this function template work ? (disambiguate the call). I don't think we ever talked about "operator templates" (or if it's a thing in c++).

✎ Last edited 9 months ago by Asicx

👍 0 ➡ Reply



Alex

Author

➡ Reply to Asicx¹⁰ 🕒 September 20, 2024 12:19 pm PDT

This function template will literally match everything you try to print. The ambiguous error happens because when T is a built-in type, the compiler doesn't know whether to use this function or the the std::ostream function to print a built-in type.

Four options come to mind:

1. Specialize the template for each type (but then you might as well just write separate functions).
2. Constrain what types this template will accept to just Apple and Banana (e.g. using concepts).
3. Use a non-operator function with a distinct name since there will be no conflict with built-ins.
4. Do the simplest thing and just use separate overloads.

👍 4 ➡ Reply



Asicx

➡ Reply to Alex¹¹ 🕒 September 21, 2024 8:17 am PDT

"Do the simplest thing and just use separate overloads." :D. You are right, but thanks to this question and you, i just learned about concepts and it worked !

```

1 // Define a concept that restricts T to be either Apple or Banana
2 template <typename T>
3 concept Fruits = std::is_same_v<T, Apple> || std::is_same_v<T, Banana>;
4
5 // Overload operator<< for types that satisfy the Fruits concept
6 template <Fruits T>
7 std::ostream& operator<<(std::ostream& out, const T& myClass)
8 {
9
10     if constexpr (std::is_same_v<T, Apple>)
11     {
12         out << "Apple(" << myClass.getName() << ", " <<
myClass.getColor() << ", " << myClass.getFiber() << ')';
13     }
14     else if constexpr (std::is_same_v<T, Banana>)
15     {
16         out << "Banana(" << myClass.getName() << ", " <<
myClass.getColor() << ')';
17     }
18
19     return out;
20 }

```

Lesson idea: concepts.

👍 4 ➡ Reply



Muhammad Altaaf

🗨 Reply to [Asicx](#)¹² ⌚ February 12, 2025 11:36 am PST

Thank you for the code!

👍 1 ➡ Reply



Alex Author

🗨 Reply to [Asicx](#)¹² ⌚ September 23, 2024 7:48 pm PDT

Very cool!

Concepts are definitely on my todo, but I won't get to them for a while.

👍 5 ➡ Reply



sambor

⌚ August 24, 2024 11:02 am PDT

in the quiz solution in the fruit class there are 2 getters:

```

1 | const std::string& getName() const { return m_name; }
2 | const std::string& getColor() const { return m_color; }

```

wouldnt it be a better choice to return a std::string instead of a const string reference?

👍 0 ➡ Reply

**Alex**

Author

Reply to [sambor](#)¹³ August 26, 2024 12:12 am PDT

No. That would return a `std::string` by value, which would require making a copy of the member each time the function was called.

Returning a const reference allows the caller to have read-only access to the member without making a copy.

👍 0

Reply

**sambor**Reply to [Alex](#)¹⁴ August 26, 2024 12:44 am PDT

Oh sorry, i meant `std::string_view` not `std::string`

👍 0

Reply

**Alex**

Author

Reply to [sambor](#)¹⁵ August 30, 2024 9:24 am PDT

Still no. That requires converting a `std::string` into a `std::string_view` every time the getter is called. Better to return a reference to the actual member and let the caller convert it to a `std::string_view` if that's what they want.

👍 4

Reply

**Serena**

May 6, 2024 9:13 am PDT

```
const std::string& type() const { return m_type; }
```

In this function, the first `const` says "I return a constant value. It's read-only.", and the second `const` says "I don't change any class members. `const` objects can call me.", right?

Should getters always use this format? I thought only the 2nd `const` was recommended... Or is this a string-reference kind of thing?

👍 0

Reply

**Serena**Reply to [Serena](#)¹⁶ May 6, 2024 9:23 am PDT

oh.. nvm. after re-reading [member functions returning references page](https://www.learncpp.com/cpp-tutorial/member-functions-returning-references-to-data-members/) (https://www.learncpp.com/cpp-tutorial/member-functions-returning-references-to-data-members/) ¹⁷ (again...) i think it's a reference thing...

👍 0

Reply

**Henry Bushnell**

January 27, 2024 11:58 am PST

Dear Alex,

In an effort to reduce repeated code, I tried to cast the apple back into a fruit for the overloaded `operator<<` and it worked! However I am worried that this could possibly be dangerous for some reason, such as not defining exactly what to do with the extra `m_fiber` data. Might this be the case?

EDIT: This is the method suggested in Lesson 24.7 for others who might have the same question. :)

EDIT II: Would using the method in Lesson 25.11 be a better solution?

```
1 class Apple : public Fruit
2 {
3 private:
4     double m_fiber{};
5
6 public:
7     Apple(std::string_view name, std::string_view color, double fiber)
8         : Fruit{name, color}
9         , m_fiber { fiber }
10    {
11    }
12
13    friend std::ostream& operator<<(std::ostream& out, const Apple& apple)
14    {
15        return out << "Apple(" << static_cast<Fruit>(apple) << ", " << apple.m_fiber
16        << ")";
17    }
18 };
```

 Last edited 1 year ago by Henry Bushnell

 0  Reply



Alex Author

 Reply to [Henry Bushnell](#) ¹⁸  January 31, 2024 6:26 pm PST

Casting a Derived object into a Base object could be dangerous, as this slices off the Derived portion of the object. We cover slicing in lesson <https://www.learncpp.com/cpp-tutorial/object-slicing/>.

It's usually better to cast a Derived object into a Base&. This Base& will be treated like a Base, but doesn't cause the Derived portion to be sliced off.

 0  Reply



Zoltan

 October 26, 2023 5:29 am PDT

Another W lesson.

And another one of the lessons where I only really got the point when I did the exercise, that's when it clicked in.

 0  Reply



Timo

🕒 October 1, 2023 5:06 am PDT

"Here's what actually happens when derived is instantiated:

Memory for derived is set aside (enough for both the Base and Derived portions)

The appropriate **Derived constructor** is called

The Base object is constructed first using the appropriate **Base constructor**. If no base constructor is specified, the default constructor will be used.

The member initializer list initializes variables

The body of the constructor executes

Control is returned to the caller"

In the previous chapter you showed us that first the base constructor is called and then the derived constructor. Why is this now the opposite?

✍️ Last edited 1 year ago by Timo

👍 0

↩️ Reply



Alex

Author

🗨️ Reply to Timo¹⁹ 🕒 October 2, 2023 1:18 pm PDT

A Derived constructor must be invoked to construct a Derived object. However, the Derived constructor first calls the Base constructor to construct the Base portion, and then control returns back to the Derived constructor to finish constructing the Derived portion.

If I said something that contradicts this elsewhere, please let me know so I can correct it.

👍 0

↩️ Reply



Timo

🗨️ Reply to Alex²⁰ 🕒 October 2, 2023 11:58 pm PDT

I did understand it now. Thanks for explanation. This was a better explanation ;)

👍 0

↩️ Reply



...

🕒 September 9, 2023 2:43 pm PDT

in the quiz isn't better using `string_view` as return type for `getName()` and `getColor()` because the variables will not be destroyed and the `std::string_view` will not dangling, instead of a `const std::string&`?

👍 1

↩️ Reply



Alex

Author

🗨️ Reply to ...²¹ 🕒 September 12, 2023 5:19 pm PDT

No. The `m_name` and `m_color` members have type `std::string`. If we return a reference of type `const std::string&`, then no conversion is required -- we can access these members via the reference. If we return a `std::string_view`, then the compiler must create and return a temporary `std::string_view` every time the function is called. This is less efficient.

This is covered in lesson <https://www.learncpp.com/cpp-tutorial/access-functions/>

 Last edited 1 year ago by Alex

 2  Reply



Serena

 Reply to [Alex](#) ²²  May 6, 2024 9:04 am PDT

I scanned that lesson, but the results for `std::string&` vs `std::string_view` for getter return values where here:

<https://www.learncpp.com/cpp-tutorial/member-functions-returning-references-to-data-members/>

 Last edited 1 year ago by Serena

 0  Reply



Alex

Author

 Reply to [Serena](#) ²³  May 8, 2024 1:16 pm PDT

Thanks. This got refactored since the initial comment was posted.

 Last edited 1 year ago by Alex

 1  Reply



...

 Reply to [Alex](#) ²²  September 12, 2023 10:50 pm PDT

Oh thx

 0  Reply

Links

1. <https://www.learncpp.com/author/Alex/>
2. <https://www.learncpp.com/cpp-tutorial/virtual-destructors-virtual-assignment-and-overriding-virtualization/>
3. `javascript:void(0)`
4. <https://www.learncpp.com/cpp-tutorial/inheritance-and-access-specifiers/>

5. <https://www.learncpp.com/>
6. <https://www.learncpp.com/cpp-tutorial/order-of-construction-of-derived-classes/>
7. <https://www.learncpp.com/constructors-and-initialization-of-derived-classes/>
8. <https://www.learncpp.com/wordpress/alexsthreadedcomments/>
9. <https://gravatar.com/>
10. <https://www.learncpp.com/cpp-tutorial/constructors-and-initialization-of-derived-classes/#comment-602042>
11. <https://www.learncpp.com/cpp-tutorial/constructors-and-initialization-of-derived-classes/#comment-602162>
12. <https://www.learncpp.com/cpp-tutorial/constructors-and-initialization-of-derived-classes/#comment-602184>
13. <https://www.learncpp.com/cpp-tutorial/constructors-and-initialization-of-derived-classes/#comment-601221>
14. <https://www.learncpp.com/cpp-tutorial/constructors-and-initialization-of-derived-classes/#comment-601294>
15. <https://www.learncpp.com/cpp-tutorial/constructors-and-initialization-of-derived-classes/#comment-601296>
16. <https://www.learncpp.com/cpp-tutorial/constructors-and-initialization-of-derived-classes/#comment-596745>
17. <https://www.learncpp.com/cpp-tutorial/member-functions-returning-references-to-data-members/>
18. <https://www.learncpp.com/cpp-tutorial/constructors-and-initialization-of-derived-classes/#comment-592964>
19. <https://www.learncpp.com/cpp-tutorial/constructors-and-initialization-of-derived-classes/#comment-588014>
20. <https://www.learncpp.com/cpp-tutorial/constructors-and-initialization-of-derived-classes/#comment-588113>
21. <https://www.learncpp.com/cpp-tutorial/constructors-and-initialization-of-derived-classes/#comment-586854>
22. <https://www.learncpp.com/cpp-tutorial/constructors-and-initialization-of-derived-classes/#comment-587082>
23. <https://www.learncpp.com/cpp-tutorial/constructors-and-initialization-of-derived-classes/#comment-596744>
24. <https://g.ezoic.net/privacy/learncpp.com>