

## 14.10 — Constructor member initializer lists

👤 ALEX<sup>1</sup> ⌚ FEBRUARY 5, 2025

This lesson continues our introduction of constructors from lesson [14.9 -- Introduction to constructors](#) (<https://www.learncpp.com/cpp-tutorial/introduction-to-constructors/>)<sup>2</sup>.

### Member initialization via a member initialization list

To have a constructor initialize members, we do so using a **member initializer list** (often called a “member initialization list”). Do not confuse this with the similarly named “initializer list” that is used to initialize aggregates with a list of values.

Member initialization lists are something that is best learned by example. In the following example, our `Foo(int, int)` constructor has been updated to use a member initializer list to initialize `m_x`, and `m_y`:

```
1  #include <iostream>
2
3  class Foo
4  {
5  private:
6      int m_x {};
7      int m_y {};
8
9  public:
10     Foo(int x, int y)
11         : m_x { x }, m_y { y } // here's our member initialization list
12     {
13         std::cout << "Foo(" << x << ", " << y << ") constructed\n";
14     }
15
16     void print() const
17     {
18         std::cout << "Foo(" << m_x << ", " << m_y << ")\n";
19     }
20 };
21
22 int main()
23 {
24     Foo foo{ 6, 7 };
25     foo.print();
26
27     return 0;
28 }
```

The member initializer list is defined after the constructor parameters. It begins with a colon (:), and then lists each member to initialize along with the initialization value for that variable, separated by a comma. You must use a direct form of initialization here (preferably using braces, but parentheses works as well) -- using copy initialization (with an equals) does not work here. Also note that the member initializer list does not end in a semicolon.

This program produces the following output:

```
Foo(6, 7) constructed
Foo(6, 7)
```

When `foo` is instantiated, the members in the initialization list are initialized with the specified initialization values. In this case, the member initializer list initializes `m_x` to the value of `x` (which is `6`), and `m_y` to the value of `y` (which is `7`). Then the body of the constructor runs.

When the `print()` member function is called, you can see that `m_x` still has value `6` and `m_y` still has value `7`.

## Member initializer list formatting

C++ provides a lot of freedom to format your member initializer lists as you prefer, as it doesn't care where you put your colon, commas, or whitespace.

The following styles are all valid (and you're likely to see all three in practice):

```
1 | Foo(int x, int y) : m_x { x }, m_y { y }
2 | {
3 | }
```

```
1 | Foo(int x, int y) :
2 |     m_x { x },
3 |     m_y { y }
4 | {
5 | }
```

```
1 | Foo(int x, int y)
2 |     : m_x { x }
3 |     , m_y { y }
4 | {
5 | }
```

Our recommendation is to use the third style above:

- Put the colon on the line after the constructor name, as this cleanly separates the member initializer list from the function prototype.
- Indent your member initializer list, to make it easier to see the function names.

If the member initialization list is short/trivial, all initializers can go on one line:

```
1 | Foo(int x, int y)
2 |     : m_x { x }, m_y { y }
3 | {
4 | }
```

Otherwise (or if you prefer), each member and initializer pair can be placed on a separate line (starting with a comma to maintain alignment):

```
1 Foo(int x, int y)
2   : m_x { x }
3   , m_y { y }
4   {
5   }
```

## Member initialization order

Because the C++ standard says so, the members in a member initializer list are always initialized in the order in which they are defined inside the class (not in the order they are defined in the member initializer list).

In the above example, because `m_x` is defined before `m_y` in the class definition, `m_x` will be initialized first (even if it is not listed first in the member initializer list).

Because we intuitively expect variables to be initialized left to right, this can cause subtle errors to occur. Consider the following example:

```
1 #include <algorithm> // for std::max
2 #include <iostream>
3
4 class Foo
5 {
6 private:
7     int m_x{};
8     int m_y{};
9
10 public:
11     Foo(int x, int y)
12         : m_y { std::max(x, y) }, m_x { m_y } // issue on this line
13     {
14     }
15
16     void print() const
17     {
18         std::cout << "Foo(" << m_x << ", " << m_y << ")\n";
19     }
20 };
21
22 int main()
23 {
24     Foo foo { 6, 7 };
25     foo.print();
26
27     return 0;
28 }
```

In the above example, our intent is to calculate the larger of the initialization values passed in (via `std::max(x, y)`) and then use this value to initialize both `m_x` and `m_y`. However, on the author's machine, the following result is printed:

```
Foo( -858993460, 7)
```

What happened? Even though `m_y` is listed first in the member initialization list, because `m_x` is defined first in the class, `m_x` gets initialized first. And `m_x` gets initialized to the value of `m_y`, which hasn't been initialized yet. Finally, `m_y` gets initialized to the greater of the initialization values.

To help prevent such errors, members in the member initializer list should be listed in the order in which they are defined in the class. Some compilers will issue a warning if members are initialized out of order.

## Best practice

Member variables in a member initializer list should be listed in order that they are defined in the class.

It's also a good idea to avoid initializing members using the value of other members (if possible). That way, even if you do make a mistake in the initialization order, it shouldn't matter because there are no dependencies between initialization values.

## Member initializer list vs default member initializers

Members can be initialized in a few different ways:

- If a member is listed in the member initializer list, that initialization value is used
- Otherwise, if the member has a default member initializer, that initialization value is used
- Otherwise, the member is default-initialized.

This means that if a member has both a default member initializer and is listed in the member initializer list for the constructor, the member initializer list value takes precedence.

Here's an example showing all three initialization methods:

```
1  #include <iostream>
2
3  class Foo
4  {
5  private:
6      int m_x {};    // default member initializer (will be ignored)
7      int m_y { 2 }; // default member initializer (will be used)
8      int m_z;       // no initializer
9
10 public:
11     Foo(int x)
12         : m_x { x } // member initializer list
13     {
14         std::cout << "Foo constructed\n";
15     }
16
17     void print() const
18     {
19         std::cout << "Foo(" << m_x << ", " << m_y << ", " << m_z << ")\n";
20     }
21 };
22
23 int main()
24 {
25     Foo foo { 6 };
26     foo.print();
27
28     return 0;
29 }
```

On the author's machine, this output:

```
Foo constructed
```

```
Foo(6, 2, -858993460)
```

Here's what's happening. When `foo` is constructed, only `m_x` appears in the member initializer list, so `m_x` is first initialized to `6`. `m_y` is not in the member initialization list, but it does have a default member initializer, so it is initialized to `2`. `m_z` is neither in the member initialization list, nor does it have a default member initializer, so it is default-initialized (which for fundamental types, means it is left uninitialized). Thus, when we print the value of `m_z`, we get undefined behavior.

## Constructor function bodies

The bodies of constructors functions are most often left empty. This is because we primarily use constructor for initialization, which is done via the member initializer list. If that is all we need to do, then we don't need any statements in the body of the constructor.

However, because the statements in the body of the constructor execute after the member initializer list has executed, we can add statements to do any other setup tasks required. In the above examples, we print something to the console to show that the constructor executed, but we could do other things like open a file or database, allocate memory, etc...

New programmers sometimes use the body of the constructor to assign values to members:

```
1  #include <iostream>
2
3  class Foo
4  {
5  private:
6      int m_x { 0 };
7      int m_y { 1 };
8
9  public:
10     Foo(int x, int y)
11     {
12         m_x = x; // incorrect: this is an assignment, not an initialization
13         m_y = y; // incorrect: this is an assignment, not an initialization
14     }
15
16     void print() const
17     {
18         std::cout << "Foo(" << m_x << ", " << m_y << ")\n";
19     }
20 };
21
22 int main()
23 {
24     Foo foo { 6, 7 };
25     foo.print();
26
27     return 0;
28 }
```

Although in this simple case this will produce the expected result, in case where members are required to be initialized (such as for data members that are const or references) assignment will not work.

### Key insight

Once the member initializer list has finished executing, the object is considered initialized. Once the function body has finished executing, the object is considered constructed.

## Best practice

Prefer using the member initializer list to initialize your members over assigning values in the body of the constructor.

## Detecting and handling invalid arguments to constructors

Consider the following Fraction class:

```
1 class Fraction
2 {
3 private:
4     int m_numerator {};
5     int m_denominator {};
6
7 public:
8     Fraction(int numerator, int denominator):
9         m_numerator { numerator }, m_denominator { denominator }
10    {
11    }
12 };
```

Because a Fraction is a numerator divided by a denominator, the denominator of a fraction cannot be zero (otherwise we get a divide by zero, which is mathematically undefined). In other words, this class has an invariant that `m_denominator` cannot be `0`.

## Related content

We discussed class invariants in lesson [14.2 -- Introduction to classes](https://www.learncpp.com/cpp-tutorial/introduction-to-classes/) (<https://www.learncpp.com/cpp-tutorial/introduction-to-classes/>)<sup>3</sup>.

So what do we do when the user tries to create a Fraction with a zero denominator (e.g. `Fraction f { 1, 0 };`)?

Inside a member initializer list, our tools for detecting and handling errors are quite limited. We can use the conditional operator to detect an error, but then what?

```
1 class Fraction
2 {
3 private:
4     int m_numerator {};
5     int m_denominator {};
6
7 public:
8     Fraction(int numerator, int denominator):
9         m_numerator { numerator }, m_denominator { denominator != 0.0 ? denominator :
10    ??? } // what do we do here?
11    {
12    }
13 };
```

We could change the denominator to a valid value, but then the user is going to get a `Fraction` that doesn't contain the values they asked for, and we don't have any way to notify them that we did something unexpected. Thus, we typically won't try to do any kind of validation in the member initializer list -- we'll just initialize the members with the values passed in, and then try to deal with the situation.

Inside the body of the constructor, we can use statements, so we have more options for detecting and handling errors. This is a good place to `assert` or `static_assert` that the arguments passed in are semantically valid, but that doesn't actually handle runtime errors in a production build.

When a constructor cannot construct a semantically valid object, we say it has failed.

---

## When constructors fail (a prelude)

In lesson [9.4 -- Detecting and handling errors](https://www.learncpp.com/cpp-tutorial/detecting-and-handling-errors/) (<https://www.learncpp.com/cpp-tutorial/detecting-and-handling-errors/>)<sup>4</sup>, we introduced the topic of error handling, and discussed some options for handling cases where a function cannot proceed due to an error occurring. Since constructors are functions, they are susceptible to the same issues.

In that lesson, we suggested 4 strategies for dealing with such errors:

- Resolve the error within the function.
- Pass the error back to the caller to deal with.
- Halt the program.
- Throw an exception.

In most cases, we don't have enough information to resolve such issues entirely within the constructor. So fixing the issue is generally not an option.

With non-member and non-special member functions, we can pass an error back to the caller to deal with. But constructors have no return value, so we don't have a good way to do that. In some cases, we can add an `isValid()` member function (or an overloaded conversion to `bool`) that returns whether the object is currently in a valid state or not. For example, an `isValid()` function for `Fraction` would return `true` when `m_denominator != 0.0`. But this means the caller has to remember to actually call the function any time a new `Fraction` is created. And having semantically invalid objects that are accessible is likely to lead to bugs. So while this is better than nothing, it's not that great of an option.

In certain types of programs, we can just halt the entire program and let the user rerun the program with the proper inputs... but in most cases, that's just not acceptable. So probably not.

And that leaves throwing an exception. Exceptions abort the construction process entirely, which means the user never gets access to a semantically invalid object. So in most cases, throwing an exception is the best thing to do in these situations.

### Key insight

Throwing an exception is usually the best thing to do when a constructor fails (and cannot recover). We discuss this further in lessons [27.5 -- Exceptions, classes, and inheritance](https://www.learncpp.com/cpp-tutorial/exceptions-classes-and-inheritance/) (<https://www.learncpp.com/cpp-tutorial/exceptions-classes-and-inheritance/>)<sup>5</sup> and [27.7 -- Function try blocks](https://www.learncpp.com/cpp-tutorial/function-try-blocks/) (<https://www.learncpp.com/cpp-tutorial/function-try-blocks/>)<sup>6</sup>.

### Author's note

For now, we'll generally assume that construction of our class object succeeds in creating a semantically valid object.

## For advanced readers

If exceptions aren't possible or desired (either because you've decided not to use them or because you haven't learned about them yet), there is one other reasonable option. Instead of letting the user create the class directly, provide a function that either returns an instance of the class or something that indicates failure.

In the following example, our `createFraction()` function returns a `std::optional<Fraction>` that optionally contains a valid `Fraction`. If it does, then we can use that `Fraction`. If not, then the caller can detect that and deal with it. We cover `std::optional` in lesson [12.15 -- std::optional](https://www.learncpp.com/cpp-tutorial/stdoptional/) (<https://www.learncpp.com/cpp-tutorial/stdoptional/>)<sup>7</sup> and friend functions in lesson [15.8 -- Friend non-member functions](https://www.learncpp.com/cpp-tutorial/friend-non-member-functions/) (<https://www.learncpp.com/cpp-tutorial/friend-non-member-functions/>)<sup>8</sup>.

```
1  #include <iostream>
2  #include <optional>
3
4  class Fraction
5  {
6  private:
7      int m_numerator { 0 };
8      int m_denominator { 1 };
9
10     // private constructor can't be called by public
11     Fraction(int numerator, int denominator):
12         m_numerator { numerator }, m_denominator { denominator }
13     {
14     }
15
16 public:
17     // Allow this function to access private members
18     friend std::optional<Fraction> createFraction(int numerator, int denominator);
19 };
20
21 std::optional<Fraction> createFraction(int numerator, int denominator)
22 {
23     if (denominator == 0)
24         return {};
25
26     return Fraction{numerator, denominator};
27 }
28
29 int main()
30 {
31     auto f1 { createFraction(0, 1) };
32     if (f1)
33     {
34         std::cout << "Fraction created\n";
35     }
36
37     auto f2 { createFraction(0, 0) };
38     if (!f2)
39     {
40         std::cout << "Bad fraction\n";
41     }
42 }
```



# Quiz time

## Question #1

Write a class named Ball. Ball should have two private member variables, one to hold a color, and one to hold a radius. Also write a function to print out the color and radius of the ball.

The following sample program should compile:

```
1 | int main()
2 | {
3 |     Ball blue { "blue", 10.0 };
4 |     print(blue);
5 |
6 |     Ball red { "red", 12.0 };
7 |     print(red);
8 |
9 |     return 0;
10| }
```

and produce the result:

```
Ball(blue, 10)
Ball(red, 12)
```

[Show Solution](#) (javascript:void(0))<sup>9</sup>

## Question #2

Why did we make `print()` a non-member function instead of a member function?

[Show Solution](#) (javascript:void(0))<sup>9</sup>

## Question #3

Why did we make `m_color` a `std::string` instead of a `std::string_view`?

[Show Solution](#) (javascript:void(0))<sup>9</sup>



**Next lesson**

**14.11** [Default constructors and default arguments](#)

12



**Back to table of contents**

13



**Previous lesson**

**14.9** [Introduction to constructors](#)

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

Leave a comment...



Name\*



@ Email\*



Notify me about replies:

**POST COMMENT**

Find a mistake? Leave a comment above!?

 Avatars from <https://gravatar.com/><sup>17</sup> are connected to your provided email address.

464 COMMENTS

Newest ▼

**Copernicus**

May 28, 2025 12:26 am PDT

Question #1

```

1  #include <iostream>
2  #include <string>
3
4  class Ball
5  {
6  public:
7      Ball(std::string colour, double radius)
8          : m_colour{ colour }, m_radius{ radius }
9      {
10     }
11
12     const std::string& getColour() const
13     {
14         return m_colour;
15     }
16
17     double getRadius() const
18     {
19         return m_radius;
20     }
21
22 private:
23     std::string m_colour{"unknown"};
24     double m_radius{ 0.0 };
25 };
26
27 static void display(const Ball& ball)
28 {
29     std::cout << "Ball(" << ball.getColour() << ", " << ball.getRadius() << ")\n";
30 }
31
32 int main()
33 {
34     Ball blue{ "blue", 10.0 };
35     display(blue);
36
37     Ball red{ "red", 12.0 };
38     display(red);
39
40     return 0;
41 }

```

## Question #2

Per best practice.

## Question #3

Because we want our classes to be owners. `std::string_view` is a viewer not an owner.

👍 0    ➡ Reply



**LapTQ**

🕒 February 3, 2025 6:57 am PST

Can you check for a typo here:

| Otherwise, the member is default "initialized".

Should it be "un"initialized?

👍 1    ➡ Reply



Alex

Author

Reply to [LapTQ](#)<sup>18</sup> ⌚ February 5, 2025 6:26 pm PST

Updated lesson to use "default-initialized" to make clear I am talking about default-initialization.

👍 0

➡ Reply



loser 27yo

⌚ February 3, 2025 1:47 am PST

```

1  #include <iostream>
2  #include <string>
3  #include <string_view>
4
5  enum class Color
6  {
7      white,
8      blue,
9      red,
10     green,
11 };
12
13 class Ball
14 {
15 public:
16     Ball(Color c, double r) : m_color{c}, m_radius{r}
17     { }
18
19     void setBallColor(Color c)
20     {
21         m_color = c;
22     }
23     std::string_view getColor() const
24     {
25         switch (m_color)
26         {
27             case Color::white:
28                 return "white";
29             case Color::blue:
30                 return "blue";
31             case Color::red:
32                 return "red";
33             case Color::green:
34                 return "green";
35             default:
36                 return "unknown";
37         }
38     }
39
40     void setBallRadius(double r)
41     {
42         m_radius = r;
43     }
44     double getRadius() const { return m_radius; }
45
46 private:
47     Color m_color{};
48     double m_radius{};
49 };
50
51 void printBall(Ball b)
52 {
53     std::cout << "Ball (" << b.getColor() << ", " << b.getRadius() << " )\n";
54 }
55
56 int main()
57 {
58     Ball myBall{ Color::white, 6.7 };
59     printBall(myBall);
60
61     std::cout << "Radius of my ball is " << myBall.getRadius() << '\n';
62
63     Ball otherDudesBall{ Color::green, 7.1 };
64     printBall(otherDudesBall);
65
66     return 0;
67 }

```

How bad this is?

 0    Reply



**Richard**

 Reply to [loser 27yo](#)<sup>19</sup>    February 11, 2025 4:09 am PST

Not bad at all bro, keep it up!

As an extra idea, I'd move the Color enum inside the public specifier of the Ball class in case it's the only class where you'd use colors, but idk if it's a good practice or not. Anyways, good luck!

 Last edited 4 months ago by Richard

 1    Reply



**rasp**

 January 31, 2025 6:06 am PST

thanks for the lessons Alex!

 Last edited 4 months ago by rasp

 0    Reply



**Theresa Fidalgo**

 January 19, 2025 4:04 pm PST

seeing that green Next lesson button is like a dopamine hit lmao

 13    Reply

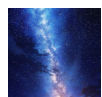


**Richard**

 Reply to [Theresa Fidalgo](#)<sup>20</sup>    February 11, 2025 4:15 am PST

real

 0    Reply



**Delici0us\_**

 January 9, 2025 1:57 am PST

Would this maybe be an interesting approach on fraction? Or is trying to get close worse than just handling it another way?

```

1  class Fraction
2  {
3  private:
4      double m_numerator {};
5      double m_denominator {};
6
7  public:
8      Fraction(double numerator, double denominator):
9          m_numerator { numerator }, m_denominator { denominator != 0.0 ? denominator :
10 std::numeric_limits<double>::min() }
11      {
12      }
13  };

```

👍 0    ➡ Reply



**Alex**

Author

🔗 Reply to DeliciOus\_ <sup>21</sup>    🕒 January 21, 2025 8:13 pm PST

A denominator of `0` more than likely means the user made an error. Given that, it's probably better to just fail than mask the issue and wait for it to manifest somewhere else. If the user really wanted a gigantic fraction, they can pass in `std::numeric_limits<double>::min()` as the denominator.

Exceptions are probably the best approach for failing here, but we haven't covered those yet.

👍 1    ➡ Reply



**NordicCat**

🕒 January 7, 2025 10:07 pm PST

what if we want to store object members dynamically (in heap) then how should we use constructor for that purpose(for example string name, I want to store this in heap instead of stack)?

👍 0    ➡ Reply



**Alex**

Author

🔗 Reply to NordicCat <sup>22</sup>    🕒 January 21, 2025 11:11 am PST

Dynamic allocation is discussed in chapter 19. <https://www.learncpp.com/cpp-tutorial/shallow-vs-deep-copying/> may be more relevant (do note this lesson hasn't been updated in a long time, so it may not be up to speed with modern best practices).

👍 0    ➡ Reply



**KLAP**

🕒 January 2, 2025 4:30 pm PST

I was trying to initialize an object with the default value by using `null` empty braces and print it but it didn't work I ended up asking chatGPT which they gave me a solution to add `Ball() = default;` in the public section. I think it would be helpful to have a small section showing ways to initialize an object with default value.

```

1  #include <iostream>
2  #include <string>
3  #include <string_view>
4
5  class Ball
6  {
7  private:
8      std::string m_color{"none"};
9      double m_radius{0.0};
10
11 public:
12     Ball() = default;
13
14     Ball(const std::string& color, double radius) :
15         m_color{color}, m_radius{radius}
16     {
17     }
18
19     const std::string& getColor() const
20     {
21         return m_color;
22     }
23
24     const double getRadius() const
25     {
26         return m_radius;
27     }
28 };
29
30 void print(Ball& b)
31 {
32     std::cout << "Ball(" << b.getColor() << ", " << b.getRadius() << ")\n";
33 }
34
35 int main()
36 {
37     Ball defaultBall;
38     print(defaultBall);
39
40     Ball blue{ "blue", 10.0 };
41     print(blue);
42
43     Ball red{ "red", 12.0 };
44     print(red);
45
46     return 0;
47 }



```

 Last edited 5 months ago by KLAP

 0  Reply



KLAP

 Reply to [KLAP](#)<sup>23</sup>  January 2, 2025 4:36 pm PST

NVM it was showed next lesson LOL.

 0  Reply



Jake

 December 26, 2024 10:52 am PST



Would there be any point of passing by reference into a constructor or setter?

Doesn't a copy of it get made regardless to initialize/set the member, making it useless?


In this specific case, wouldn't color be converted into an `std::string` from `std::string_view` anyway to have a value for `m_color`? Maybe I'm misunderstanding and `string_view` is very different from a reference or maybe I'm misunderstanding in general.

 Last edited 6 months ago by Jake

 0  Reply



**Alex** Author

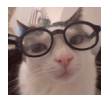
 Reply to [Jake](#)<sup>24</sup>  January 3, 2025 4:42 pm PST

If we pass an object by value into a constructor and then use that object to initialize a member, the object gets copied twice -- once into the parameter, and once into the member.

If we pass an object by reference into a constructor and then use that object to initialize a member, the object gets copied once into the member.

`std::string_view` is cheap to copy so it doesn't need to be passed by reference.

 0  Reply



**NordicCat**

 December 21, 2024 9:13 pm PST

What should we put inside the constructor's body then?

 0  Reply



**Alex** Author

 Reply to [NordicCat](#)<sup>25</sup>  December 30, 2024 12:04 am PST

I'll let Kuni take this one: <https://www.youtube.com/watch?v=647ikNd3jSI>

Very occasionally you'll find something that needs to go there because it's too awkward to put in the member initializer list, or because you need to do some exception handling.

 1  Reply

## Links

1. <https://www.learncpp.com/author/Alex/>
2. <https://www.learncpp.com/cpp-tutorial/introduction-to-constructors/>
3. <https://www.learncpp.com/cpp-tutorial/introduction-to-classes/>

4. <https://www.learncpp.com/cpp-tutorial/detecting-and-handling-errors/>
5. <https://www.learncpp.com/cpp-tutorial/exceptions-classes-and-inheritance/>
6. <https://www.learncpp.com/cpp-tutorial/function-try-blocks/>
7. <https://www.learncpp.com/cpp-tutorial/stdoptional/>
8. <https://www.learncpp.com/cpp-tutorial/friend-non-member-functions/>
9. `javascript:void(0)`
10. <https://www.learncpp.com/cpp-tutorial/the-benefits-of-data-hiding-encapsulation/#prefer-non-member-functions>
11. <https://www.learncpp.com/cpp-tutorial/struct-miscellany/>
12. <https://www.learncpp.com/cpp-tutorial/default-constructors-and-default-arguments/>
13. <https://www.learncpp.com/>
14. <https://www.learncpp.com/constructor-member-initializer-lists/>
15. <https://www.learncpp.com/site-news/learncppcom-tutorials-now-with-syntax-highlighting/>
16. <https://www.learncpp.com/cpp-tutorial/using-an-integrated-debugger-stepping/>
17. <https://gravatar.com/>
18. <https://www.learncpp.com/cpp-tutorial/constructor-member-initializer-lists/#comment-607345>
19. <https://www.learncpp.com/cpp-tutorial/constructor-member-initializer-lists/#comment-607333>
20. <https://www.learncpp.com/cpp-tutorial/constructor-member-initializer-lists/#comment-606760>
21. <https://www.learncpp.com/cpp-tutorial/constructor-member-initializer-lists/#comment-606395>
22. <https://www.learncpp.com/cpp-tutorial/constructor-member-initializer-lists/#comment-606370>
23. <https://www.learncpp.com/cpp-tutorial/constructor-member-initializer-lists/#comment-606119>
24. <https://www.learncpp.com/cpp-tutorial/constructor-member-initializer-lists/#comment-605743>
25. <https://www.learncpp.com/cpp-tutorial/constructor-member-initializer-lists/#comment-605565>
26. <https://g.ezoic.net/privacy/learncpp.com>