



2.5 — Introduction to local scope

👤 ALEX¹ ⌚ MAY 21, 2022

Local variables

Function parameters, as well as variables defined inside the function body, are called **local variables** (as opposed to global variables, which we'll discuss in a future chapter).

```
1 | int add(int x, int y) // function parameters x and y are local variables
2 | {
3 |     int z{ x + y }; // z is a local variable too
4 |
5 |     return z;
6 | }
```

In this lesson, we'll take a look at some properties of local variables in more detail.

Local variable lifetime

In lesson [1.3 -- Introduction to objects and variables](https://www.learncpp.com/cpp-tutorial/introduction-to-objects-and-variables/) (<https://www.learncpp.com/cpp-tutorial/introduction-to-objects-and-variables/>)², we discussed how a variable definition such as `int x;` causes the variable to be instantiated (created) when this statement is executed. Function parameters are created and initialized when the function is entered, and variables within the function body are created and initialized at the point of definition.

For example:

```
1 | int add(int x, int y) // x and y created and initialized here
2 | {
3 |     int z{ x + y }; // z created and initialized here
4 |
5 |     return z;
6 | }
```

The natural follow-up question is, “so when is an instantiated variable destroyed?”. Local variables are destroyed in the opposite order of creation at the end of the set of curly braces in which it is defined (or for a function parameter, at the end of the function).

```

1 | int add(int x, int y)
2 | {
3 |     int z{ x + y };
4 |
5 |     return z;
6 | } // z, y, and x destroyed here

```

Much like a person's lifetime is defined to be the time between their birth and death, an object's **lifetime** is defined to be the time between its creation and destruction. Note that variable creation and destruction happen when the program is running (called runtime), not at compile time. Therefore, lifetime is a runtime property.

For advanced readers

The above rules around creation, initialization, and destruction are guarantees. That is, objects must be created and initialized no later than the point of definition, and destroyed no earlier than the end of the set of the curly braces in which they are defined (or, for function parameters, at the end of the function).

In actuality, the C++ specification gives compilers a lot of flexibility to determine when local variables are created and destroyed. Objects may be created earlier, or destroyed later for optimization purposes. Most often, local variables are created when the function is entered, and destroyed in the opposite order of creation when the function is exited. We'll discuss this in more detail in a future lesson, when we talk about the call stack.

Here's a slightly more complex program demonstrating the lifetime of a variable named *x*:

```

1 | #include <iostream>
2 |
3 | void doSomething()
4 | {
5 |     std::cout << "Hello!\n";
6 | }
7 |
8 | int main()
9 | {
10 |     int x{ 0 }; // x's lifetime begins here
11 |
12 |     doSomething(); // x is still alive during this function call
13 |
14 |     return 0;
15 | } // x's lifetime ends here

```

In the above program, *x*'s lifetime runs from the point of definition to the end of function *main*. This includes the time spent during the execution of function *doSomething*.

Local scope

An identifier's **scope** determines where the identifier can be accessed within the source code. When an identifier can be accessed, we say it is **in scope**. When an identifier can not be accessed, we say it is **out of scope**. Scope is a compile-time property, and trying to use an identifier when it is not in scope will result in a compile error.

A local variable's scope begins at the point of variable definition, and stops at the end of the set of curly braces in which it is defined (or for function parameters, at the end of the function). This ensures variables can not be used before the point of definition (even if the compiler opts to create them before then). Local variables defined in one function are also not in scope in other functions that are called.

Here's a program demonstrating the scope of a variable named `x`:

```
1  #include <iostream>
2
3  // x is not in scope anywhere in this function
4  void doSomething()
5  {
6      std::cout << "Hello!\n";
7  }
8
9  int main()
10 {
11     // x can not be used here because it's not in scope yet
12
13     int x{ 0 }; // x enters scope here and can now be used within this function
14
15     doSomething();
16
17     return 0;
18 } // x goes out of scope here and can no longer be used
```

In the above program, variable `x` enters scope at the point of definition and goes out of scope at the end of the `main` function. Note that variable `x` is not in scope anywhere inside of function `doSomething()`. The fact that function `main` calls function `doSomething` is irrelevant in this context.

“Out of scope” vs “going out of scope”

The terms “out of scope” and “going out of scope” can be confusing to new programmers.

An identifier is “out of scope” anywhere it can not be accessed within the code. In the example above, the identifier `x` is in-scope from its point of definition to the end of the `main()` function. The identifier is out-of-scope outside of that code region.

The term “going out of scope” is typically applied to objects rather than identifiers. We say an object “goes out of scope” at the end of the scope (the end curly brace) in which the object was instantiated. In the example above, the object named `x` “goes out of scope” at the end of the function `main()`.

A local variable's lifetime ends at the point where it "goes out of scope", so local variables are destroyed at this point.

Note that not all types of variables are destroyed when they "go out of scope". We'll see examples of these in future lessons.

Another example

Here's a slightly more complex example. Remember, lifetime is a runtime property, and scope is a compile-time property, so although we are talking about both in the same program, they are enforced at different points.

```
1  #include <iostream>
2
3  int add(int x, int y) // x and y are created and enter scope here
4  {
5      // x and y are visible/usable within this function only
6      return x + y;
7  } // y and x go out of scope and are destroyed here
8
9  int main()
10 {
11     int a{ 5 }; // a is created, initialized, and enters scope here
12     int b{ 6 }; // b is created, initialized, and enters scope here
13
14     // a and b are usable within this function only
15     std::cout << add(a, b) << '\n'; // calls function add() with x=5 and y=6
16
17     return 0;
18 } // b and a go out of scope and are destroyed here
```

Parameters *x* and *y* are created when the *add* function is called, can only be seen/used within function *add*, and are destroyed at the end of *add*. Variables *a* and *b* are created within function *main*, can only be seen/used within function *main*, and are destroyed at the end of *main*.

To enhance your understanding of how all this fits together, let's trace through this program in a little more detail. The following happens, in order:

- execution starts at the top of *main*
- *main*'s variable *a* is created and given value 5
- *main*'s variable *b* is created and given value 6
- function *add* is called with values 5 and 6 for arguments
- *add*'s variable *x* is created and initialized with value 5
- *add*'s variable *y* is created and initialized with value 6
- *operator+* evaluates expression *x + y* to produce the value 11
- *add* copies the value 11 back to caller *main*
- *add*'s *y* and *x* are destroyed
- *main* prints 11 to the console
- *main* returns 0 to the operating system

- *main's* *b* and *a* are destroyed

And we're done.

Note that if function *add* were to be called twice, parameters *x* and *y* would be created and destroyed twice -- once for each call. In a program with lots of functions and function calls, variables are created and destroyed often.

Functional separation

In the above example, it's easy to see that variables *a* and *b* are different variables from *x* and *y*.

Now consider the following similar program:

```
1  #include <iostream>
2
3  int add(int x, int y) // add's x and y are created and enter scope here
4  {
5      // add's x and y are visible/usable within this function only
6      return x + y;
7  } // add's y and x go out of scope and are destroyed here
8
9  int main()
10 {
11     int x{ 5 }; // main's x is created, initialized, and enters scope here
12     int y{ 6 }; // main's y is created, initialized, and enters scope here
13
14     // main's x and y are usable within this function only
15     std::cout << add(x, y) << '\n'; // calls function add() with x=5 and y=6
16
17     return 0;
18 } // main's y and x go out of scope and are destroyed here
```

In this example, all we've done is change the names of variables *a* and *b* inside of function *main* to *x* and *y*. This program compiles and runs identically, even though functions *main* and *add* both have variables named *x* and *y*. Why does this work?

First, we need to recognize that even though functions *main* and *add* both have variables named *x* and *y*, these variables are distinct. The *x* and *y* in function *main* have nothing to do with the *x* and *y* in function *add* -- they just happen to share the same names.

Second, when inside of function *main*, the names *x* and *y* refer to *main's* locally scoped variables *x* and *y*. Those variables can only be seen (and used) inside of *main*. Similarly, when inside function *add*, the names *x* and *y* refer to function parameters *x* and *y*, which can only be seen (and used) inside of *add*.

In short, neither *add* nor *main* know that the other function has variables with the same names. Because the scopes don't overlap, it's always clear to the compiler which *x* and *y* are being referred to at any time.

Key insight

Names used for function parameters or variables declared in a function body are only visible within the function that declares them. This means local variables within a function can be named without regard for the names of variables in other functions. This helps keep functions independent.

We'll talk more about local scope, and other kinds of scope, in a future chapter.

Where to define local variables

Local variables inside the function body should be defined as close to their first use as reasonable:

```
1  #include <iostream>
2
3  int main()
4  {
5      std::cout << "Enter an integer: ";
6      int x{}; // x defined here
7      std::cin >> x; // and used here
8
9      std::cout << "Enter another integer: ";
10     int y{}; // y defined here
11     std::cin >> y; // and used here
12
13     int sum{ x + y }; // sum defined here
14     std::cout << "The sum is: " << sum << '\n'; // and used
15     here
16
17     return 0;
18 }
```

In the above example, each variable is defined just before it is first used. There's no need to be strict about this -- if you prefer to swap lines 5 and 6, that's fine.

Best practice

Define your local variables as close to their first use as reasonable.

Quiz time

Question #1

What does the following program print?

```

1  #include <iostream>
2
3  void doIt(int x)
4  {
5      int y{ 4 };
6      std::cout << "doIt: x = " << x << " y = " << y << '\n';
7
8      x = 3;
9      std::cout << "doIt: x = " << x << " y = " << y << '\n';
10 }
11
12 int main()
13 {
14     int x{ 1 };
15     int y{ 2 };
16
17     std::cout << "main: x = " << x << " y = " << y << '\n';
18
19     doIt(x);
20
21     std::cout << "main: x = " << x << " y = " << y << '\n';
22
23     return 0;
24 }

```

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



Next lesson

2.6 [Why functions are useful, and how to use them effectively](#)

4



Back to table of contents

5



Previous lesson

2.4 [Introduction to function parameters and arguments](#)



6

7

Leave a comment...

 Name*

 Email* 

 Find a mistake? Leave a comment! 

Notify me about replies:



POST COMMENT

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

276 COMMENTS

Newest ▼



hantao

🕒 October 29, 2022 11:38 pm

Excellent tutorial. I am still confused about "identifiers in/out of scope" vs "objects going out of scope" and "runtime property" vs "compliance property".

How will the differences affect our program? Or they are just terminology?

👍 0

↩ Reply



Question on run-time

🕒 September 4, 2022 1:20 pm

>>we discussed how a variable definition such as `int x;` causes the variable to be instantiated (created) when this statement is executed. Function parameters are created and initialized when the function is entered, and variables within the function body are created and initialized at the point of definition.

Does it mean all the variable creation and initialization happening at run-time either they are defined as function parameters or defined inside a function body?

👍 0

↩ Reply



Alex Author

👤 Reply to [Question on run-time](#) 10 🕒 September 12, 2022 2:00 pm

 Reply to [Question on run-time](#) · September 13, 2022 3:00 pm

No. Later in this series, we discuss global variables, which are defined outside a function body, but can still be initialized at runtime (when the program starts).

 2  Reply

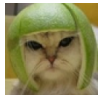


Austin

🕒 August 13, 2022 6:38 pm

So in the "Another Example" in the "Out of scope" vs "going out of scope", is `a` and `b` entirely different than `x` and `y`? Does the `return x+y` do anything in that situation? Or does `x` and `y` somehow magically become `a` and `b`, and it just works?

 0  Reply



Alex Author

 Reply to [Austin](#) ¹¹ · August 15, 2022 11:30 am

Yes, `a` and `b` are separate variables than `x` and `y`. When function `add(x, y)` is called, variable `a` is created and initialized with the value of `x` and `b` is created and initialized with the value of `y`. This doesn't affect `x` and `y`.

The return `x+y` just returns a value back to the caller, which it uses to print. It's not particularly relevant to the example.

 1  Reply



Paul

🕒 July 21, 2022 10:13 am

```
#include <iostream>
```

```
int doAddition(int x, int y) //Does addition
```

```
{  
    int sum{};  
    sum = x + y;
```

```
    return sum;  
}
```

```
int main()  
{  
    std::cout << "enter an integer ";  
    int num1;
```

```
std::cin >> num1;
std::cout << "enter another integer ";
int num2;
std::cin >> num2;

std::cout << doAddition(num1, num2) << '\n';

return 0;
}
```

//First Attempt, any thoughts?

👍 0

➡ Reply



tenchu

👤 Reply to [Paul](#)¹² ⌚ September 1, 2022 10:32 am

Nothing really wrong with your code, good job!

But there is room for improvement, no need to make a variable for something you will just return.

```
int doAddition(int x, int y)
{
    return x + y;
}
```

Also remember DRY(Do not repeat yourself)? You could make a function to get the input, of course you can remove the std::cout outside the function if you really want "enter another integer". the "input" variable only exists inside this function, so it doesn't matter it has "the same name".

```
int getValue()
{
    std::cout << "Enter a integer: ";
    int input{};
    std::cin >> input;

    return input;
}
```

Then you can call it in main like this:

```
int main()
{
    std::cout << doAddition(getValue(), getValue());

    return 0;
}
```



1



Reply

**Vince**

🕒 May 19, 2022 6:04 pm

Thanks for the tutorial!

I just noticed in this sentence under the **Local Scope** section: "A local variable's scope begins at the point of variable definition, and stops at the end of the set of curly braces in which they are defined (or for function parameters, at the end of the function)", should the part "in which they are defined" be "in which it is defined", since you're talking about a variable, instead of variables?



0



Reply

**Matin**

🕒 April 21, 2022 10:35 am

Hey, first, thanks for an awesome tutorial <3.

Second, if variables are alive in "main" function ,during calling other functions, is there anyway to pass the variable itself from "main" function into other functions' argument instead of copying the variable' data?



1



Reply

**Alex**

Author

Reply to [Matin](#)¹³

🕒 April 21, 2022 11:48 am

Not exactly, but the same effect can be achieved by using "pass by reference" instead of "pass by value". This is covered in lesson <https://www.learncpp.com/cpp-tutorial/pass-by-lvalue-reference/>



1



Reply

**Phw**

🕒 April 1, 2022 8:24 pm

I'm trying to calculate sum of the 2 dimensional array d, but when i passed the variable

1 | `sumd`

into the function call it did nothing to it and the printed result is 0. I declared sumd in advance so i think the problem isn't with going out of scope. Please help me answer this

```
1  #include <iostream>
2  using namespace std;
3
4  void sumArrAllEntry (int arr[][5], int T, int S, int sum)
5  {
6      for (int i{}; i < T; ++i)
7          {
8              for (int j{}; j < S; ++j)
9                  sum += arr[j][i];
10         }
11     }
12
13     int main()
14     {
15         const int S{4}, T{5}; // declare S and T
16         int d[S][T]{0, 7, 0, 8, 0},
17             {0, 7, 0, 8, 10},
18             {6, 0, 10, 3, 10},
19             {6, 0, 15, 0, 10}};
20         int K{20}; //declare K
21         int sumd{0};
22
23         sumArrAllEntry (d, T, S, sumd);
24         cout << sumd;
25
26         return 0;
27     }
```

👍 0

➡ Reply



Azhar Khan

🗨 Reply to Phw¹⁴ ⌚ April 9, 2022 2:06 am

You didn't return value of sum from sumArrAllEntry function and assign it to sumd. That's why it kept printing sumd's original value which was 0.

```

1  #include <iostream>
2  using namespace std;
3
4  int sumArrAllEntry (int arr[][5], int T, int S, int sum)
5  {
6      for (int i{}; i < T; ++i)
7      {
8          for (int j{}; j < S; ++j)
9              sum += arr[j][i];
10     }
11
12     return sum;
13 }
14
15 int main()
16 {
17     const int S{4}, T{5}; // declare S and T
18     int d[S][T]{0, 7, 0, 8, 0},
19         {0, 7, 0, 8, 10},
20         {6, 0, 10, 3, 10},
21         {6, 0, 15, 0, 10}};
22     int K{20}; //declare K
23     int sumd{0};
24
25     sumd=sumArrAllEntry (d, T, S, sumd);
26     cout << sumd;
27
28     return 0;
29 }

```

👍 0

➡ Reply



Phw

🗨 Reply to [Azhar Khan](#)¹⁵ 🕒 April 10, 2022 8:17 pm

Thank you, but i thought passing sumd at `sumArrAllEntry (d, T, S, sumd);` is enough to change it? I did the same with an array as the sum variable and it worked. I don't really get it haha

👍 1

➡ Reply



Landen

🗨 Reply to [Phw](#)¹⁶ 🕒 April 14, 2022 1:29 am

Using your original program, alter line 4 to be:

```

1 | void sumArrAllEntry (int arr[][5], int T, int S, int &sum)

```

Adding an ampersand (&) between **int** and **sum** allows you to directly change **sumd** that's defined in **main**. This is called **pass by reference**. You will then see an output of **100**. **Pass by reference**, to put it in very basic terms (since it isn't covered yet), allows you to use **sum** as an alias for **sumd** in the function **sumArrAllEntry**.

👍 2 ➡ Reply



Phw

🗨 Reply to [Landen](#)¹⁷ ⌚ April 14, 2022 2:28 am

Thank you, I studied pass by reference already but didn't think to use it! That'll definitely change the variable

👍 0 ➡ Reply



Azhar Khan

🗨 Reply to [Phw](#)¹⁶ ⌚ April 12, 2022 11:39 am

It works fine with arrays for some reason but for normal variables we will have to learn pointers. Keep learning :)

👍 0 ➡ Reply



Chris Offner

⌚ March 19, 2022 2:32 pm

The sentence

> People often misuse the term “going out of scope” to mean
in the Tip seems to be cut off.

👍 0 ➡ Reply



Alex Author

🗨 Reply to [Chris Offner](#)¹⁸ ⌚ March 20, 2022 7:15 pm

Converted this into a section and fixed. Thanks for pointing this out.

👍 1 ➡ Reply



Arjan

⌚ March 4, 2022 5:35 am

Here is something I noticed when playing around with what I learnt in this lesson. If you add a function to a `std::cout` statement, its return statement will be printed. But be aware! If there were any `std::cout` statements INSIDE the function, those will also be printed! Run this piece of code.

```
1  #include <iostream>
2
3  int doSomething()
4  {
5      std::cout << 4 << '\n';
6
7      return 5;
8  }
9
10 int main()
11 {
12     std::cout << "doSomething() is executed.\n";
13     doSomething();
14     std::cout << '\n';
15
16     std::cout << "doSomething() is executed AND the return value is printed.\n";
17     std::cout << doSomething() << "\n";
18
19     return 0;
20 }
```

The output will be:

```
1  doSomething() is executed.
2  4
3
4  doSomething() is executed AND the return value is printed.
5  4
6  5
```

👍 4 ➡ Reply



rob405

🗨 Reply to [Arjan](#)¹⁹ 🕒 March 21, 2022 4:09 pm

I think this is because the result of the function (the return value) is like a normal variable.

In other words what is happening in `cout<<doSomething()` is:

1. Execute everything before return value.
2. Return value becomes a normal value like in `cout<<3` or `cout<<"text"`
3. Return value gets printed by cout

👍 1 ➡ Reply



Catreece

Reply to [rob405](#)²⁰ April 8, 2022 8:38 am

It's weird, but looking at it, it makes sense.

In line 13, it doesn't ask for anything to be returned from `doSomething()`, but in `doSomething()` it prints off the 4 on line 5.

In line 17, it's saying to run `doSomething()` and whatever its return result happens to be is to be injected into `cout` to be printed, and it happens to print the 4 off while running `doSomething()` before it returns the value to `main()`.

As such, line 13 never gets told to use `doSomething()`'s output, but line 17 does use that return value.

👍 1 ➡ Reply



Mins273

March 1, 2022 4:07 am

```
int add(int x, int y) // x and y created and initialized here
{
    int z{ x + y }; // z created and initialized here

    return z;
}
```

It says that the variables `x` and `y` are initialized in the function `add()`, shouldn't it be defined instead since it isn't assigned a value until it receives the value from the function calling it? I know this is just a minor thing and won't affect anything but I just want to clarify the basics.

I just tried a simple function

```
#include <iostream>

int main(int x)
{
    std::cout << x;

    return 0;
}
```

and when I executed the program, to my surprise, it returned a value of 1, I expected it to return 0 assuming the if it was initialized it would mean it was blank initialized like `int x{}` since I did not assign `x` a value...

Then I changed the code a little


```
#include <iostream>
```

```
int main()
```

```
{
```

```
int x;
```

```
std::cout << x;
```

```
return 0;
```

```
}
```

but this time like I expected x wasn't assigned a value and there was a compile error message.

👍 0

➡ Reply



Mins273

👤 Reply to [Mins273](#) ²¹ ⌚ March 1, 2022 4:54 am

*EDIT

```
int add(int x, int y) // x and y created and initialized here
```

```
{
```

```
int z{ x + y }; // z created and initialized here
```

```
return z;
```

```
}
```

It says that the variables x and y are initialized in the function add(), shouldn't it be defined instead since it isn't assigned a value until it receives the value from the function calling it? I know this is just a minor thing and won't affect anything but I just want to clarify the basics.

I just tried a simple function

```
#include <iostream>
```

```
int main(int x)
```

```
{
```

```
std::cout << x;
```

```
return 0;
```

```
}
```

and when I executed the program, to my surprise, it returned a value of 1, I expected it to return 0 assuming the if it was initialized it would mean it was blank initialized like `int x{}` since I did not assign x a value...

so my question here is why did it return 1 even though a value wasn't assigned to it?

Then I changed the code a little

```
#include <iostream>
```

```
int main()
{
int x;
std::cout << x;

return 0;
}
```

but this time like I expected x wasn't assigned a value and there was a compile error message.

so another question here is why is it different when I initialize variable x in the main() function compared to in the function body itself even though they are written the same way, just different places?

👍 0 ➡ Reply



BK87

👤 Reply to [Mins273](#) ²² ⌚ March 10, 2022 7:18 pm

Hi Mins273,

I assume that this will be discussed later in the course, but to help clarify what is going on here, your program:

```
1 | #include <iostream>
2 |
3 | int main(int x)
4 | {
5 |     std::cout << x;
6 |
7 |     return 0;
8 | }
```

This prints 1 at the command line because there is an extended syntax for the main() function that also allows calling the program with input parameters from the command line which has a standard idiom like:

```
1 | int main(int argc, char** argv)
2 | {
3 |     return 0;
4 | }
```

The input parameters argc and argv are the number of input arguments (argc stands for "argument count") and an array of character arrays (argv stands for "argument values", and the type char** probably will make more sense later when pointers and arrays are explained). By convention, argc will always have a value of at least 1 and the first value in

argv will be a string or char array giving the absolute path to the program. So, what happened in your case with:

```
1 | #include <iostream>
2 |
3 | int main(int x)
4 | {
5 |     std::cout << x;
6 |
7 |     return 0;
8 | }
```

is that the input argument x was initialized with a value of 1 because it took the place of the standard first argument (normally called argc, but as the lesson mentioned, the names of the input parameters to a function are arbitrary). Your program then printed the value 1 to the terminal and exited successfully (returning 0 back to the operating system).

Cheers,
Brandon

 0  Reply

Links

1. <https://www.learncpp.com/author/Alex/>
2. <https://www.learncpp.com/cpp-tutorial/introduction-to-objects-and-variables/>
3. [javascript:void\(0\)](#)
4. <https://www.learncpp.com/cpp-tutorial/why-functions-are-useful-and-how-to-use-them-effectively/>
5. <https://www.learncpp.com/>
6. <https://www.learncpp.com/cpp-tutorial/introduction-to-function-parameters-and-arguments/>
7. <https://www.learncpp.com/introduction-to-local-scope/>
8. <https://www.learncpp.com/cpp-tutorial/void/>
9. <https://gravatar.com/>
10. <https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-572712>
11. <https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-572041>
12. <https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-570914>
13. <https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-567889>
14. <https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-567140>
15. <https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-567399>

16. <https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-567439>
17. <https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-567588>
18. <https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-566650>
19. <https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-565972>
20. <https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-566756>
21. <https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-565853>
22. <https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-565855>
23. <https://www.ezoic.com/what-is-ezoic/>