2.4 — Introduction to local scope

BY ALEX ON FEBRUARY 8TH, 2015 | LAST MODIFIED BY NASCARDRIVER ON FEBRUARY 7TH, 2020

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).

```
int add(int x, int y) // function parameters x and y are local variables
{
    int z{ x + y }; // z is a local variable too

    return z;
}
```

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 variables**, 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:

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

    return z;
}
```

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).

```
int add(int x, int y)
{
    int z{ x + y };

    return z;
} // 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*:

```
#include <iostream>

void doSomething()
{
    std::cout << "Hello!\n";
}

int main()
{
    int x{ 0 }; // x's lifetime begins here

    doSomething(); // x is still alive during this function call

    return 0;
} // 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 they are 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).

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

```
#include <iostream>

// x is not in scope anywhere in this function
void doSomething()
{
    std::cout << "Hello!\n";
}

int main()
{
    // x can not be used here because it's not in scope yet

    int x{ 0 }; // x enters scope here and can now be used

    doSomething();

    return 0;
} // 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.

Note that local variables have the same definitions for scope and lifetime. For local variables, scope and lifetime are linked -- that is, a variable's lifetime starts when it enters scope, and ends when it goes out of scope.

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.

```
#include <iostream>

int add(int x, int y) // x and y are created and enter scope here
{
    // x and y are visible/usable within this function only
    return x + y;
} // y and x go out of scope and are destroyed here

int main()
{
    int a{ 5 }; // a is created, initialized, and enters scope here
    int b{ 6 }; // b is created, initialized, and enters scope here

    // a and b are usable within this function only
    std::cout << add(a, b) << '\n'; // calls function add() with x=5 and y=6

    return 0;
} // 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:

```
#include <iostream>

int add(int x, int y) // add's x and y are created and enter scope here
{
    // add's x and y are visible/usable within this function only
    return x + y;
} // add's y and x go out of scope and are destroyed here

int main()
{
    int x{ 5 }; // main's x is created, initialized, and enters scope here
    int y{ 6 }; // main's y is created, initialized, and enters scope here

    // main's x and y are usable within this function only
    std::cout << add(x, y) << '\n'; // calls function add() with x=5 and y=6

    return 0;
} // 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:

```
#include <iostream>

int main()
{
    std::cout << "Enter an integer: ";
    int x{}; // x defined here
    std::cin >> x; // and used here

    std::cout << "Enter another integer: ";
    int y{}; // y defined here
```

```
    std::cin >> y; // and used here

    int sum{ x + y }; // sum defined here
    std::cout << "The sum is: " << sum << '\n'; // and used here

    return 0;
}
```

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?

```
#include <iostream>

void doIt(int x)
{
    int y{ 4 };
    std::cout << "doIt: x = " << x << " y = " << y << '\n';

    x = 3;
    std::cout << "doIt: x = " << x << " y = " << y << '\n';
}

int main()
{
    int x{ 1 };
    int y{ 2 };

    std::cout << "main: x = " << x << " y = " << y << '\n';

    doIt(x);

    std::cout << "main: x = " << x << " y = " << y << '\n';

    return 0;
}
```

Show Solution

2.5 -- Why functions are useful, and how to use them effectively

Index

2.3 -- Introduction to function parameters and arguments

C++ TUTORIAL | PRINT THIS POST

170 comments to 2.4 — Introduction to local scope

« Older Comments 1 2 3

Vitaliy Sh.
February 6, 2020 at 1:49 am · Reply

Hi Sires!

```
// in "Local variable lifetime", "For example:"
// Can your please change this snippet to remind us the variable initialization topic?
// ...
int z{ x + y }; // z created and initialized here
int a;     // is created but NOT initialized here


// colon?                          : "so...?".
"The natural follow-up question is, \“so...?\"."


// was in lesson 2.3: "A warning about function argument order of evaluation"
// standard didn't says anything about the evaluation order of the function parameters
//  z, y and x destroyed here ?
"// z, y, and x destroyed here"


// in "Where to define local variables", code example
    std::cout << "The sum is: " << sum; // and used here

// ++        std::cout << '\n';

    return 0;


// in Question #1, Solution
```

```
//                      created and
"doIt's parameter x is created initialized with value 1"
// std::cout, like above and below?
"doIt prints doIt: x = 1 y = 4"



// Also i want to propose a change to Question #1 code.
#include <iostream>

// const char* is a invitation to the next chapters ;)
void printIt(const char* caller, int x, int y)
{
    std::cout
        << caller
        << ": x = "
        << x
        << " y = "
        << y
        << '\n';
}

void doIt(int x)
{
    int y{ 4 };
    printIt("doIt", x, y);

    x = 3;
    printIt("doIt", x, y);
}

int main()
{
    int x{ 1 };
    int y{ 2 };

    printIt("main", x, y);

    doIt(x);

    printIt("main", x, y);

    return 0;
}
```

prince
January 24, 2020 at 5:49 am · Reply

In programing where does the execution starts? does it starts on the first statement inside function "main" or in the first statement in the first function?

nascardriver
January 24, 2020 at 6:02 am · Reply

It starts in the `main` function. The order of functions in the source code doesn't affect the order they run in.

sexy

November 19, 2019 at 8:40 am · Reply

```
#include <iostream>

int add(int x, int y) // add's x and y are created here
{
    return x + y;
} // add's x and y go out of scope and are destroyed here

int main()
{
    int x = 5; // main's x is created here
    int y = 6; // main's y is created here
    std::cout << add(x, y) << std::endl; // the values from main's x and y are copied into
    return 0;
}
```

i need help with this code

Alex
November 19, 2019 at 1:00 pm · Reply

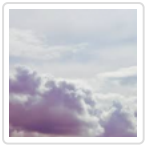
Can you be more specific about what part you find confusing?

Chayim
October 21, 2019 at 11:40 pm · Reply

In the quiz, in function doit x is initialized to 3, so why when function doit is called in main it’s x is main’s initialization 1 until it executes the second function in doit?

nascardriver
October 22, 2019 at 1:06 am · Reply

`x` in `doIt` is initialized by the caller, in this case in `main` line 19. The `x` in `main` is not the same `x` as the `x` in `doIt`, they could have different names.
In line 19, the value of `main`'s `x` is used to initialize the `x` of `doIt`.
Line 8 changes the value of `doIt`'s `x` to 1, but it doesn't affect `main`'s `x`, because that's a different variable that just happens to have the same name.



Chayim
October 22, 2019 at 9:19 am · Reply

I know that x in Main and x in doIt are not the same. My question was that doIt's x was already declared in it's own function as 3, so why when it's called in main it uses main's x and not it's own declared x as 3?
I thought myself that when main calls doIt function doIt is being executed in order by it's two phases at the first part of it's function it is not declared yet so it uses main's x declared as 1, and only at it's second phase of it's function it is declared as 3, but my question was that why is it this way, because it's declared in this function as 3 so even at the beginning of the function it should not use main's x but it's own.

nascardriver
October 23, 2019 at 2:40 am · Reply

> it's declared in this function as 3
It's not. It's declared in line 3 and will be initialized whenever `doIt` is called.

```
1 doIt(123); // x = 123
2
3 int iApple{ 99 };
4 doIt(iApple); // x = 99
```

Line 8 is an assignment. It changes the value of `x`.

Chayim
October 23, 2019 at 11:18 pm · Reply

"In this function" I meant in beginning when function doIt is declared, so why when doIt is called in main it uses main's x and not it's own?

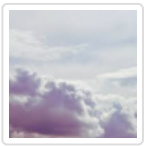
nascardriver
October 23, 2019 at 11:54 pm · Reply

That's because `main` calls it with

```
1 doIt(x); // @x is @main's @x
```

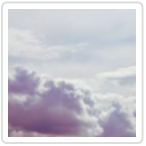
When you call a function that has a parameter, you have to pass in a value. It can't be called like

```
1 doIt(); // Illegal, we need to supply a value of @doIt's @x.
```



Chayim
November 3, 2019 at 10:29 pm · Reply

The clear answer of my question is:
because the compiler compiles the contents of code files sequentially like it's stated in chapter 2.7, so when the compiler compiles the function 'doIt' the it's not assigned yet until next section of the function that assigns it, so it uses main's x.



Chayim
November 5, 2019 at 5:12 am · Reply

The clear answer of my question is:
because the compiler compiles the contents of code files sequentially like it's stated in chapter 2.7, so when the compiler compiles the function 'doIt' it's not assigned yet until next section of the function that assigns it, so it uses main's x that was assigned before.

Dan
August 28, 2019 at 9:35 am · Reply

Quick scroll through the comments and didn't see this question. I apologize if it's a repeat. Is there a best practice when naming a parameter variable and the argument variable passed to it? Even though they can have the same name like in the example demonstrating scope, it seems to me that it could be a source of confusion to anyone reading your code if they did.

nascardriver
August 29, 2019 at 12:04 am · Reply

If they describe the same thing, they should have the same name. `x` and `y` are bad names, unless they're used for coordinates where they're well established names.
Local variables are a feature of many programming languages, anyone who knows a language will understand them.
If this is your first language, it's ok to be confused, it'll pass.

Dan
August 30, 2019 at 3:39 am · Reply

Thanks!

Jose
July 29, 2019 at 6:31 am · Reply

Hi, when explaining the quiz code at the end I think
"doIt's x and y are destroyed" and
"main's x and y are destroyed"
should have the identifiers in reverse order to keep consistency with what´s been explained previously.

Alex
July 29, 2019 at 11:27 am · Reply

Updated. Thanks for the suggestion.

Singh
June 18, 2019 at 8:46 am · Reply

I am getting below the result of running a shared program. Just curious to know what made the malfunction while running below code whereas its working fine with other two options.

OUTPUT:
"Please add both numbers.
A: 5
B: 6
Addition of both 5 and 6 is 116296576

Program Finished."

[code]
```
#include <iostream>

int addfunction(int x, int y)
{
  std::cout<<"Addition of both " <<x <<" and " <<y <<" is " <<x+y;
  /* why can't we use above where as both the belw sections works fine.
   instead of
```

```
    int c{x+y};
    return c;
    OR
    retrun x+y; */
}

int main()
{
    int a, b;
    std::cout<<"Please add both numbers.\n";
    std::cout<<"A: ";
    std::cin>>a;
    std::cout<<"B: ";
    std::cin>>b;

    std::cout<<addfunction(a,b);

    std::cout<<"\n\nProgram Finished.";

return 0;
}
[\code]
```

Arman
June 19, 2019 at 2:48 am · Reply

Mistake 1:type of addfunction function must be void.because it doesnt return any value and just print sth on screen.
Mistake 2:you shouldn't use cout<<addfunction(a,b); because addfunction doesnt return any value.you should simply call the function itself without cout.

nascardriver
June 19, 2019 at 3:17 am · Reply

Correct. Your code tags will work if you close them using a forward slash (/). You should always print a line feed as the last character of you output to avoid mixed lines.

Prateek
May 3, 2019 at 4:39 am · Reply

In second snippet(above) :

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

I am a bit confused here. I understand that 'x' and 'y' has been created but how is it initialized?
Here 'x' and 'y'has not been assigned any value.

Please do reply

nascardriver
May 3, 2019 at 4:43 am · Reply

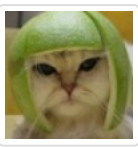
They're initialized from the values that are passed by the caller.

```
1 add(3, 1); // x = 3, y = 1
```

Cade
April 14, 2019 at 2:09 am · Reply

Under the local scope section it says: “Here’s program” instead of “Here’s a program” or “Here is a program”. Not try a be annoying just wanted to help.

Have a Good Day:)



Alex
April 14, 2019 at 8:13 am · Reply

Thanks, appreciate you pointing out the typo. Fixed!

Michael Johnston
April 1, 2019 at 1:54 am · Reply

Great explanation for people new to programming; however, as someone coming from another language where, by default, parameters are passed by reference, I got the quiz wrong. It might be helpful to include another ‘for advanced users’ snippet on this which states this explicitly.

nascardriver
April 1, 2019 at 2:50 am · Reply

Hi!
Which language is it you're talking about? I only know languages where objects or arrays are passed by reference by default, but none where all arguments are passed by reference.

Dirk de Klerk
April 3, 2019 at 10:18 am · Reply

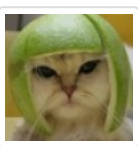
You seemed to have skipped 2.3. He explicitly states that when arguments are passed to parameters when calling functions it is "passed by value". He even wrote it in bold.

Saurabh
March 18, 2019 at 6:05 pm · Reply

https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/
This ensures variables can not be used before the point of definition (even if the compiler opts to create them before them).

I think that this should be … compiler opts to create them before then

(then instead of them)



Alex
March 19, 2019 at 8:35 pm · Reply

Fixed. Thanks!

Juan
March 3, 2019 at 10:58 am · Reply

The page is very good, I would just say: they can relax with the use of "astute readers". It sounds to differentiate groups of people, according to their qualities (which leads to discrimination). If this is a page to learn something without prior knowledge, it is not necessary to assume anything that has not been explained. Perhaps, instead of astute readers, less ambiguous explanations would be needed. That is why the work of rewriting parts of the tutorials, to update their content, is a good practice. Thanks for that hard work!

Alex
March 4, 2019 at 9:15 pm · Reply

Fair enough. All astuteness has been removed. I appreciate the feedback.

Hana
March 2, 2019 at 1:54 pm · Reply

The output of the solution's "doit" is "doIt" with a capital I instead.

Alex
March 4, 2019 at 9:08 pm · Reply

Thanks! Fixed.

Wilson
February 24, 2019 at 1:48 pm · Reply

First three code snippets has a small typo here:

int z{ x + y; }

(should be int z{ x + y };)....

otherwise, great guide, Alex. Thanks!

Alex
February 26, 2019 at 7:35 pm · Reply

Fixed! Thanks for pointing out the typo.

GG
February 5, 2019 at 4:38 am · Reply

in another example: main‘s b and z are destroyed should be.... main‘s b and a are destroyed

Alex
February 7, 2019 at 6:07 am · Reply

Thanks!

Hans

October 26, 2018 at 12:25 am · Reply

Helo there, first of all thanks for making this site. I learn so much here. I wanna ask something about the answer for this quiz. Why does the variable x in void doIt has value of 1? void doIt(int x) didn't give it value so shouldn't it become unpredictable?

nascardriver
October 26, 2018 at 1:54 am · Reply

@x is passed to @doIt from @main. You might want to re-read lesson 1.4a, because this is an important concept.

Hans
October 26, 2018 at 7:06 am · Reply

Whoa thanks nascardriver for the reply and enlightment. I understand it now. I miss the little x inside the doIt there. My bad XD

Sourabh
October 18, 2018 at 10:45 pm · Reply

HEY
if nothing is initialized in int x= {nothing is assigned here to x}
what will be the value given by compiler to X???

nascardriver
October 19, 2018 at 2:55 am · Reply

```
1 int x; // Undefined
2 int x =; // Syntax error
3 int x = 0; // 0
4 int x{ 0 }; // 0 (Uniform initialization, lesson 2.1)
5 int x{}; // 0
```

Kio
March 7, 2018 at 3:58 pm · Reply

Hi Alex,

Maybe some kind of suggestion. What do you think,
using same example below this

```
1  #include <iostream>
2
3  int add(int x, int y) // add's x and y are created here
4  {
5      return x + y;
6  } // add's x and y go out of scope and are destroyed here
7
8  int main()
9  {
10     int x = 5; // main's x is created here
11     int y = 6; // main's y is created here
```

```
12      std::cout << add(x, y) << std::endl; // the values from main's x and y are copied into
13      return 0;
14  }
```

additional print address of the variable x and y (so beginners can see, that their addresses are different). Or maybe this is confusing?

« Older Comments 1 2 3