1.4d — A first look at local scope

BY ALEX ON FEBRUARY 8TH, 2015 | LAST MODIFIED BY ALEX ON DECEMBER 13TH, 2017

You already saw in the lesson **1.3 -- a first look at variables** that when the CPU executes a statement like `int x;`, this causes the variable to be instantiated (created).

The natural follow-up question is, “so when is an instantiated variable destroyed?”

Appetite for destruction

A variable’s **scope** determines when a variable can be seen and used during the time it is instantiated. Function parameters and variables defined inside the function body both have **local scope**. That is, those variables can only be seen and used within the function that defines them. Local variables are created at the point of definition, and destroyed when they go out of scope (usually at the end of the function).

Here’s a simple example:

```cpp
#include <iostream>

int main()
{
    int x; // variable x is created here
    std::cout << "Enter a value for x: ";
    std::cin >> x;
    std::cout << "You entered: " << x << std::endl;

    return 0;
} // variable x is destroyed here
```

Here’s a slightly more complex example:

```cpp
#include <iostream>

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

int main()
{
    int a = 5; // a is created and initialized here
    int b = 6; // b is created and initialized here
    // a and b are usable within this function only
    std::cout << add(a, b) << std::endl; // calls function add() with x=a and y=b
    return 0;
} // a and b go out of scope and are destroyed here
```

In function add(), parameters x and y are created when the function is called, can only be seen/used within function add(), and are destroyed at the end of the add() function.
Variable 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, let’s trace through this program in a little more detail. The following happens, in order:

- main() is executed
- 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 given value 5
- add’s variable y is created and given value 6
- operator + adds 5 and 6 to produce the value 11

- add returns the value 11 to the caller (main)
- add’s x and y are destroyed
- main prints 11 to the console
- main returns 0 to the operating system
- main’s a and b are destroyed

And we’re done.

Note that if function add() were to be called twice, function add() 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.

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 program:

```cpp
#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 add's x and
 y
    return 0;
} // main's x and y 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 main() to x and y. This program still runs fine, even though both main() and add() both have variables named x and y. Why does this work?

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

Second, when inside of main(), the names x and y refer to the 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, and it’s always clear to the compiler which x and y are being referred to at any time. This means that functions don’t need to know or care what other functions name their variables, which is fantastic for us because it makes life easy.

We’ll talk more about local scope, and other kinds of scope, in chapter 4.

Rule: Names used for function parameters or variables declared in a function body are only visible within the function that declares them.

Quiz

1) What does the following program print?

```cpp
#include <iostream>

void doIt(int x)
{
    int y = 4;
    std::cout << "doIt: x = " << x << " y = " << y << std::endl;
    x = 3;
    std::cout << "doIt: x = " << x << " y = " << y << std::endl;
}

int main()
{
    int x = 1;
    int y = 2;
    std::cout << "main: x = " << x << " y = " << y << std::endl;
    doIt(x);
    std::cout << "main: x = " << x << " y = " << y << std::endl;
    return 0;
}
```

Quiz answers

1) **Hide Solution**

```
main: x = 1 y = 2
doit: x = 1 y = 4
doit: x = 3 y = 4
main: x = 1 y = 2
```

Here's what happens in this program:

- main() is executed
- main's variable x is created and given value 1
- main's variable y is created and given value 2
- cout prints "main: x = 1 y = 2"
- doIt() is called with argument 1
- doIt's variable x is created and given value 1
- doit's variable y is created and given value 4
- doIt prints "doit: x = 1 y = 4"
- doIt's variable x is assigned the value 3
- cout prints "doIt: x = 3 y = 4"
- doIt's x and y are destroyed
- cout prints "main: x = 1 y = 2"
- main returns 0 to the operating system
- main's x and y are destroyed

Note that even though doIt's variables x and y had their values assigned to something different than main's, main's were unaffected.

1.5 -- A first look at operators

Index

1.4c -- Keywords and naming identifiers

127 comments to 1.4d — A first look at local scope

« Older Comments 1 2 3