2.5 — Introduction to local scope

learncpp.com/cpp-tutorial/introduction-to-local-scope/

Local variables

Variables defined inside the body of a function are called **local variables** (as opposed to *global variables*, which we'll discuss in a future chapter):

```
int add(int x, int y)
{
    int z{ x + y }; // z is a local variable

    return z;
}
```

Function parameters are also generally considered to be local variables, and we will include them as such:

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

    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 objects and 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, the lifetime of `x` runs from the point of definition to the end of function `main`. This includes the time spent during the execution of function `doSomething`.

What happens when an object is destroyed?

In most cases, nothing. The destroyed object becomes invalid, and any further use of the object will result in undefined behavior. At some point after destruction, the memory used by the object will be freed up for reuse.

For advanced readers

If the object is a class type object, prior to destruction, a special function called a destructor is invoked. In many cases, the destructor does nothing, in which case no cost is incurred.

Local scope

An identifier's **scope** determines where the identifier can be seen and used within the source code. When an identifier can be seen and used, we say it is **in scope**. When an identifier can not be seen, we can not use it, and 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`:

```
#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 within this function

    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.

“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 cannot 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 `x` 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.

```
#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` variable `a` is created and given value `5`.
- `main` variable `b` is created and given value `6`.
- Function `add` is called with argument values `5` and `6`.
- `add` parameters `x` and `y` are created and initialized with values `5` and `6` respectively.
- The expression `x + y` is evaluated to produce the value `11`.
- `add` copies the value `11` back to caller `main`.
- `add` parameters `y` and `x` are destroyed.
- `main` prints `11` to the console.
- `main` returns `0` to the operating system.
- `main` variables `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

In modern C++, the best practice is that 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 can be initialized with intended value
	std::cout << "The sum is: " << sum << '\n';

	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.

As an aside…

Due to the limitations of older, more primitive compilers, the C language used to require all local variables be defined at the top of the function. The equivalent C++ program using that style would look like this:

```
#include <iostream>

int main()
{
        int x{}, y{}, sum{}; // how are these used?

        std::cout << "Enter an integer: ";
        std::cin >> x;

        std::cout << "Enter another integer: ";
        std::cin >> y;

        sum = x + y;
        std::cout << "The sum is: " << sum << '\n';

        return 0;
}
```

This style is suboptimal for several reasons:

- The intended use of these variables isn’t apparent at the point of definition. You have to scan through the entire function to determine where and how each variable is used.
- The intended initialization value may not be available at the top of the function (e.g. we can’t initialize `sum` to its intended value because we don’t know the value of `x` and `y` yet).
- There may be many lines between a variable’s initializer and its first use. If we don’t remember what value it was initialized with, we will have to scroll back to the top of the function, which is distracting.

This restriction was lifted in the C99 language standard.

Introduction to temporary objects

A **temporary object** (also sometimes called an **anonymous object**) is an unnamed object that is created by the compiler to store a value temporarily.

There are many different ways that temporary values can be created, but here’s a common one:

```
#include <iostream>

int getValueFromUser()
{
        std::cout << "Enter an integer: ";
        int input{};
        std::cin >> input;

        return input; // return the value of input back to the caller
}

int main()
{
        std::cout << getValueFromUser() << '\n'; // where does the returned value get
stored?

        return 0;
}
```

In the above program, the function `getValueFromUser()` returns the value stored in local variable `input` back to the caller. Because `input` will be destroyed at the end of the function, the caller receives a copy of the value so that it has a value it can use even after `input` is destroyed.

But where is the value that is copied back to the caller stored? We haven’t defined any variables in `main()`. The answer is that the return value is stored in a temporary object. This temporary object is then passed to `std::cout` to be printed.

Temporary objects have no scope at all (this makes sense, since scope is a property of an identifier, and temporary objects have no identifier).

Temporary objects are destroyed at the end of the full expression in which they are created. Thus the temporary object created to hold the return value of `getValueFromUser()` is destroyed after `std::cout << getValueFromUser() << '\n'` executes.

In the case where a temporary object is used to initialize a variable, the initialization happens before the destruction of the temporary.

In modern C++ (especially since C++17), the compiler has many tricks to avoid generating temporaries where previously it would have needed to. In the above example, since the return value of `getValueFromUser()` is immediately output, the compiler can skip creation and destruction of the temporary in `main()`, and use the return value of `getValueFromUser()` to directly initialize the parameter of `operator<<`.

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