7.3 — Local variables

In lesson 2.5 -- Introduction to local scope, we introduced `local variables`, which are variables that are defined inside a function (including function parameters).

It turns out that C++ actually doesn’t have a single attribute that defines a variable as being a local variable. Instead, local variables have several different properties that differentiate how these variables behave from other kinds of (non-local) variables. We’ll explore these properties in this and upcoming lessons.

In lesson 2.5 -- Introduction to local scope, we also introduced the concept of scope. An identifier’s `scope` determines where an 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 out of scope will result in a compile error.

Local variables have block scope

Local variables have **block scope**, which means they are *in scope* from their point of definition to the end of the block they are defined within.

Related content

Please review lesson 7.1 -- Compound statements (blocks) if you need a refresher on blocks.

```
int main()
{
    int i { 5 }; // i enters scope here
    double d { 4.0 }; // d enters scope here

    return 0;
} // d and i go out of scope here
```

Although function parameters are not defined inside the function body, for typical functions they can be considered to be part of the scope of the function body block.

```
int max(int x, int y) // x and y enter scope here
{
    // assign the greater of x or y to max
    int max{ (x > y) ? x : y }; // max enters scope here

    return max;
} // max, y, and x leave scope here
```

All variable names within a scope must be unique

Variable names must be unique within a given scope, otherwise any reference to the name will be ambiguous. Consider the following program:

```
void someFunction(int x)
{
    int x{}; // compilation failure due to name collision with function parameter
}

int main()
{
    return 0;
}
```

The above program doesn't compile because the variable `x` defined inside the function body and the function parameter `x` have the same name and both are in the same block scope.

Local variables have automatic storage duration

A variable's **storage duration** (usually just called **duration**) determines what rules govern when and how a variable will be created and destroyed. In most cases, a variable's storage duration directly determines its `lifetime`.

Related content

We discuss what a lifetime is in lesson 2.5 -- Introduction to local scope.

For example, local variables have **automatic storage duration**, which means they are created at the point of definition and destroyed at the end of the block they are defined in. For example:

```
int main()
{
    int i { 5 }; // i created and initialized here
    double d { 4.0 }; // d created and initialized here

    return 0;
} // d and i are destroyed here
```

For this reason, local variables are sometimes called **automatic variables**.

Local variables in nested blocks

Local variables can be defined inside nested blocks. This works identically to local variables in function body blocks:

```
int main() // outer block
{
    int x { 5 }; // x enters scope and is created here

    { // nested block
        int y { 7 }; // y enters scope and is created here
    } // y goes out of scope and is destroyed here

    // y can not be used here because it is out of scope in this block

    return 0;
} // x goes out of scope and is destroyed here
```

In the above example, variable `y` is defined inside a nested block. Its scope is limited from its point of definition to the end of the nested block, and its lifetime is the same. Because the scope of variable `y` is limited to the inner block in which it is defined, it's not accessible anywhere in the outer block.

Note that nested blocks are considered part of the scope of the outer block in which they are defined. Consequently, variables defined in the outer block *can* be seen inside a nested block:

```
#include <iostream>

int main()
{ // outer block

    int x { 5 }; // x enters scope and is created here

    { // nested block
        int y { 7 }; // y enters scope and is created here

        // x and y are both in scope here
        std::cout << x << " + " << y << " = " << x + y << '\n';
    } // y goes out of scope and is destroyed here

    // y can not be used here because it is out of scope in this block

    return 0;
} // x goes out of scope and is destroyed here
```

Local variables have no linkage

Identifiers have another property named *linkage*. An identifier's **linkage** determines whether a declaration of that same identifier in a different scope refers to the same object (or function).

Local variables have no linkage. Each declaration of an identifier with no linkage refers to a unique object or function.

For example:

```
int main()
{
    int x { 2 }; // local variable, no linkage

    {
        int x { 3 }; // this declaration of x refers to a different object than the
previous x
    }

    return 0;
}
```

Scope and linkage may seem somewhat similar. However, scope determines where declaration of a single identifier can be seen and used in the code. Linkage determines whether multiple declarations of the same identifier refer to the same object or not.

Related content

We discuss what happens when variables with the same name appear in nested blocks in lesson 7.5 -- Variable shadowing (name hiding).

Linkage isn’t very interesting in the context of local variables, but we’ll talk about it more in the next few lessons.

Variables should be defined in the most limited scope

If a variable is only used within a nested block, it should be defined inside that nested block:

```
#include <iostream>

int main()
{
    // do not define y here

    {
        // y is only used inside this block, so define it here
        int y { 5 };
        std::cout << y << '\n';
    }

    // otherwise y could still be used here, where it's not needed

    return 0;
}
```

By limiting the scope of a variable, you reduce the complexity of the program because the number of active variables is reduced. Further, it makes it easier to see where variables are used (or aren’t used). A variable defined inside a block can only be used within that block (or nested blocks). This can make the program easier to understand.

If a variable is needed in an outer block, it needs to be declared in the outer block:

```
#include <iostream>

int main()
{
    int y { 5 }; // we're declaring y here because we need it in this outer block
later

    {
        int x{};
        std::cin >> x;

        // if we declared y here, immediately before its actual first use...
        if (x == 4)
            y = 4;
    } // ... it would be destroyed here

    std::cout << y; // and we need y to exist here

    return 0;
}
```

The above example shows one of the rare cases where you may need to declare a variable well before its first use.

New developers sometimes wonder whether it's worth creating a nested block just to intentionally limit a variable's scope (and force it to go out of scope / be destroyed early). Doing so makes that variable simpler, but the overall function becomes longer and more complex as a result. The tradeoff generally isn't worth it. If creating a nested block seems useful to intentionally limit the scope of a chunk of code, that code might be better to put in a separate function instead.

Best practice

Define variables in the most limited existing scope. Avoid creating new blocks whose only purpose is to limit the scope of variables.

Quiz time

Question #1

Write a program that asks the user to enter two integers, one named `smaller`, the other named `larger`. If the user enters a smaller value for the second integer, use a block and a temporary variable to swap the smaller and larger values. Then print the values of the `smaller` and `larger` variables. Add comments to your code indicating where each variable dies. Note: When you print the values, `smaller` should hold the smaller input and `larger` the larger input, no matter which order they were entered in.

The program output should match the following:

```
Enter an integer: 4
Enter a larger integer: 2
Swapping the values
The smaller value is 2
The larger value is 4
```

Show Solution

Question #2

What's the difference between a variable's scope, duration, and lifetime? By default, what kind of scope and duration do local variables have (and what do those mean)?

Show Solution