4.1a — Local variables, scope, and duration

BY ALEX ON MARCH 23RD, 2015 | LAST MODIFIED BY ALEX ON MARCH 31ST, 2018

The following section builds on section **1.4b -- a first look at local scope**.

When discussing variables, it's useful to separate out the concepts of scope and duration. A variable's **scope** determines where a variable is accessible. A variable's **duration** determines where it is created and destroyed. The two concepts are often linked.

Variables defined inside a function are called **local variables**. Local variables have **automatic duration**, which means they are created (and initialized, if relevant) at the point of definition, and destroyed when the block they are defined in is exited. Local variables have **block scope** (also called **local scope**), which means they enter scope at the point of declaration and go out of scope at the end of the block that they are defined in.

Consider this simple function:

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

    return 0;

} // i and d go out of scope and are destroyed here
```

Because i and d were defined inside the block that defines the main function, they are both destroyed when main() is finished executing.

Variables defined inside nested blocks are destroyed as soon as the nested block ends:

```
int main() // outer block
{
    int n(5); // n created and initialized here

    { // begin nested block
        double d(4.0); // d created and initialized here
    } // d goes out of scope and is destroyed here

    // d can not be used here because it was already destroyed!

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

Variables defined inside a block can only be seen within that block. Because each function has its own block, variables in one function can not be seen from another function:

```
void someFunction()
{
    int value(4); // value defined here

    // value can be seen and used here

} // value goes out of scope and is destroyed here

int main()
{
    // value can not be seen or used inside this function.

    someFunction();

    // value still can not be seen or used inside this function.

    return 0;
}
```

This means functions can have variables or parameters with the same names as other functions. This is a good thing, because it means we don't have to worry about naming collisions between two independent functions. In the following example, both functions have variables named x and y. These variables in each function are unaware of the existence of other variables with the same name in other functions.

```
#include <iostream>

// add's x and y can only be seen/used within function add()
int add(int x, int y) // add's x and y are created here and can only be seen/used within add() after th
is point
{
    return x + y;
} // add's x and y are destroyed here

// main's x and y can only be seen/used within function main()
int main()
{
    int x = 5; // main's x is created here, and can be seen/used only within main() after this point
    int y = 6; // main's y is created here, and can be seen/used only within main() after this point

    std::cout << add(x, y) << std::endl; // the value from main's x and y are copied into add's x and y

    // We can still use main's x and y here

    return 0;
} // main's x and y are destroyed here
```

Nested blocks are considered part 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()
{ // start outer block

    int x(5);

    { // start nested block
        int y(7);
        // we can see both x and y from here
        std::cout << x << " + " << y << " = " << x + y;
    } // y destroyed here

    // y can not be used here because it was already destroyed!

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

Shadowing

Note that a variable inside a nested block can have the same name as a variable inside an outer block. When this happens, the nested variable "hides" the outer variable. This is called **name hiding** or **shadowing**.

```
#include <iostream>

int main()
{ // outer block
    int apples(5); // here's the outer block apples

    if (apples >= 5) // refers to outer block apples
    { // nested block
        int apples; // hides previous variable named apples

        // apples now refers to the nested block apples
        // the outer block apples is temporarily hidden

        apples = 10; // this assigns value 10 to nested block apples, not outer block apples
```

```
        std::cout << apples << '\n'; // print value of nested block apples
    } // nested block apples destroyed

    // apples now refers to the outer block apples again

    std::cout << apples << '\n'; // prints value of outer block apples

    return 0;
} // outer block apples destroyed
```

If you run this program, it prints:

```
10
5
```

In the above program, we first declare a variable named apples in the outer block. Then we declare a different variable (also named apples) in the nested block. When we assign value 10 to apples, we're assigning it to the nested block apples. After printing this value, nested block apples is destroyed, leaving outer block apples with its original value (5), which is then printed. This program executes the exact same as it would have if we'd named nested block apples something else (e.g. nbApples) and kept the names distinct (because outer block apples and nested block apples are distinct variables, they just share the same name).

Note that if the nested block apples had not been defined, the name apples in the nested block would still refer to the outer apples, so the assignment of value 10 to apples would have applied to the outer block apples:

```
#include <iostream>

int main()
{ // outer block
    int apples(5); // here's the outer block apples

    if (apples >= 5) // refers to outer block apples
    { // nested block
        // no inner block apples defined

        apples = 10; // this now applies to outer block apples, even though we're in an inner block

        std::cout << apples << '\n'; // print value of outer block apples
    } // outer block apples retains its value even after we leave the nested block

    std::cout << apples << '\n'; // prints value of outer block apples

    return 0;
} // outer block apples destroyed
```

The above program prints:

```
10
10
```

In both examples, outer block apples is not impacted by what happens to nested block apples. The only difference between the two programs is which apples the expression `apples = 10` applies to.

Shadowing is something that should generally be avoided, as it is quite confusing!

Rule: Avoid using nested variables with the same names as variables in an outer block.

Variables should be defined in the most limited scope possible

For example, 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;
    }

    // otherwise y could still be used here

    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. A variable defined inside a block can only be used within that block (or nested sub-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;
}
```

This is one of the rare cases where you may need to declare a variable well before its first use.

Rule: Define variables in the smallest scope and as close to the first use as possible..

Function parameters

Although function parameters are not defined inside the block belonging to the function, in most cases, they can be considered to have block scope.

```
int max(int x, int y) // x and y defined here
{
    // assign the greater of x or y to max
    int max = (x > y) ? x : y; // max defined here
    return max;
} // x, y, and max all die here
```

The exception case is for function-level exceptions, which we'll cover in a future section.

Summary

Variables defined inside functions are called local variables. These variables can only be accessed inside the block in which they are defined (including nested blocks), and they are destroyed as soon as the block ends.

Define variables in the smallest scope that they are used. If a variable is only used within a nested block, define it within the nested block.

Quiz

1) Write a program that asks the user to enter two integers, the second larger than the first. If the user entered a smaller integer for the second integer, use a block and a temporary variable to swap the smaller and larger values. Then print the value of the smaller and larger variables. Add comments to your code indicating where each variable dies.

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
```

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

Quiz solutions

1) **Hide Solution**

```
#include <iostream>

int main()
{
    std::cout << "Enter an integer: ";
    int smaller;
    std::cin >> smaller;

    std::cout << "Enter a larger integer: ";
    int larger;
    std::cin >> larger;

    // if user did it wrong
    if (smaller > larger)
    {
        // swap values of smaller and larger
        std::cout << "Swapping the values\n";

        int temp = larger;
        larger = smaller;
        smaller = temp;
    } // temp dies here

    std::cout << "The smaller value is: " << smaller << "\n";
    std::cout << "The larger value is: " << larger << "\n";

    return 0;
} // smaller and larger die here
```

2) **Hide Solution**

A variable's scope determines where the variable is accessible. Duration defines when a variable is created and destroyed.

Local variables have block scope, which means they can be accessed only during the block in which they are defined.

Local variables have automatic duration, which means they are created at the point of definition, and destroyed at the end of the block in which they are defined.

 **4.2 -- Global variables and linkage**

 **Index**