7.5 — Variable shadowing (name hiding)

learncpp.com/cpp-tutorial/variable-shadowing-name-hiding/

Each block defines its own scope region. So what happens when we have a variable inside a nested block that has the same name as a variable in an outer block? When this happens, the nested variable “hides” the outer variable in areas where they are both in scope. This is called **name hiding** or **shadowing**.

Shadowing of local variables

```
#include <iostream>

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

    { // nested block
        // apples refers to outer block apples here
        std::cout << apples << '\n'; // print value of outer block apples

        int apples{ 0 }; // define apples in the scope of the nested block

        // 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


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

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

If you run this program, it prints:

```
5
10
5
```

In the above program, we first declare a variable named `apples` in the outer block. This variable is visible within the inner block, which we can see by printing its value (`5`). Then we declare a different variable (also named `apples`) in the nested block. From this point to the

end of the block, the name `apples` refers to the nested block `apples`, not the outer block `apples`.

Thus, when we assign value `10` to `apples`, we're assigning it to the nested block `apples`. After printing this value (`10`), the nested block ends and nested block `apples` is destroyed. The existence and value of outer block `apples` is not affected, and we prove this by printing the value of outer block `apples` (`5`).

Note that if the nested block `apples` had not been defined, the name `apples` in the nested block would still refer to the outer block `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

    { // nested block
        // apples refers to outer block apples here
        std::cout << apples << '\n'; // print value of outer block apples

        // no inner block apples defined in this example

        apples = 10; // this applies to outer block apples

        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:

```
5
10
10
```

When inside the nested block, there's no way to directly access the shadowed variable from the outer block.

Shadowing of global variables

Similar to how variables in a nested block can shadow variables in an outer block, local variables with the same name as a global variable will shadow the global variable wherever the local variable is in scope:

```
#include <iostream>
int value { 5 }; // global variable

void foo()
{
    std::cout << "global variable value: " << value << '\n'; // value is not shadowed
here, so this refers to the global value
}

int main()
{
    int value { 7 }; // hides the global variable value until the end of this block

    ++value; // increments local value, not global value

    std::cout << "local variable value: " << value << '\n';

    foo();

    return 0;
} // local value is destroyed
```

This code prints:

```
local variable value: 8
global variable value: 5
```

However, because global variables are part of the global namespace, we can use the scope operator (::) with no prefix to tell the compiler we mean the global variable instead of the local variable.

```
#include <iostream>
int value { 5 }; // global variable

int main()
{
    int value { 7 }; // hides the global variable value
    ++value; // increments local value, not global value

    --(::value); // decrements global value, not local value (parenthesis added for
readability)

    std::cout << "local variable value: " << value << '\n';
    std::cout << "global variable value: " << ::value << '\n';

    return 0;
} // local value is destroyed
```

This code prints:

```
local variable value: 8
global variable value: 4
```

Avoid variable shadowing

Shadowing of local variables should generally be avoided, as it can lead to inadvertent errors where the wrong variable is used or modified. Some compilers will issue a warning when a variable is shadowed.

For the same reason that we recommend avoiding shadowing local variables, we recommend avoiding shadowing global variables as well. This is trivially avoidable if all of your global names use a “g_” prefix.

Best practice

Avoid variable shadowing.

For GCC/G++ users

GCC and Clang support the flag `-Wshadow` that will generate warnings if a variable is shadowed. There are several subvariants of this flag (`-Wshadow=global`, `-Wshadow=local`, and `-Wshadow=compatible-local`. Consult the GCC documentation for an explanation of the differences.

Visual Studio has such warnings enabled by default.