7.5 — Switch fallthrough and scoping

ALEX JUNE 19, 2021

This lesson continues our exploration of switch statements that we started in the prior lesson 7.4 -- Switch statement basics. In the prior lesson, we mentioned that each set of statements underneath a label should end in a `break statement` or a `return statement`.

In this lesson, we'll explore why, and talk about some switch scoping issues that sometimes trip up new programmers.

Fallthrough

When a switch expression matches a case label or optional default label, execution begins at the first statement following the matching label. Execution will then continue sequentially until one of the following termination conditions happens:

1. The end of the switch block is reached.
2. Another control flow statement (typically a `break` or `return`) causes the switch block or function to exit.
3. Something else interrupts the normal flow of the program (e.g. the OS shuts the program down, the universe implodes, etc...)

Note that the presence of another case label is *not* one of these terminating conditions -- thus, without a `break` or `return`, execution will overflow into subsequent cases.

Here is a program that exhibits this behavior:

```cpp
#include <iostream>

int main()
{
    switch (2)
    {
    case 1: // Does not match
        std::cout << 1 << '\n'; // Skipped
    case 2: // Match!
        std::cout << 2 << '\n'; // Execution begins here
    case 3:
        std::cout << 3 << '\n'; // This is also executed
    case 4:
        std::cout << 4 << '\n'; // This is also executed
    default:
        std::cout << 5 << '\n'; // This is also executed
    }

    return 0;
}
```

This program outputs the following:

```
2
3
4
5
```

This is probably not what we wanted! When execution flows from a statement underneath a label into statements underneath a subsequent label, this is called **fallthrough**.

> **Warning**
>
> Once the statements underneath a case or default label have started executing, they will overflow (fallthrough) into subsequent cases. `Break` or `return` statements are typically used to prevent this.

Since fallthrough is rarely desired or intentional, many compilers and code analysis tools will flag fallthrough as a warning.

The [[fallthrough]] attribute

Commenting intentional fallthrough is a common convention to tell other developers that fallthrough is intended. While this works for other developers, the compiler and code analysis tools don't know how to interpret comments, so it won't get rid of the warnings.

To help address this, C++17 adds a new attribute called `[[fallthrough]]`.

Attributes are a modern C++ feature that allows the programmer to provide the compiler with some additional data about the code. To specify an attribute, the attribute name is placed between double hard braces. Attributes are not statements -- rather, they can be used almost anywhere where they are contextually relevant.

The `[[fallthrough]]` attribute modifies a `null statement` to indicate that fallthrough is intentional (and no warnings should be triggered):

```cpp
#include <iostream>

int main()
{
    switch (2)
    {
    case 1:
        std::cout << 1 << '\n';
        break;
    case 2:
        std::cout << 2 << '\n'; // Execution begins here
        [[fallthrough]]; // intentional fallthrough -- note the semicolon to indicate the null statement
    case 3:
        std::cout << 3 << '\n'; // This is also executed
        break;
    }

    return 0;
}
```

This program prints:

```
2
3
```

And it should not generate any warnings about the fallthrough.

Best practice

Use the `[[fallthrough]]` attribute (along with a null statement) to indicate intentional fallthrough.

Sequential case labels

You can use the logical OR operator to combine multiple tests into a single statement:

```cpp
bool isVowel(char c)
{
    return (c=='a' || c=='e' || c=='i' || c=='o' || c=='u'
||
        c=='A' || c=='E' || c=='I' || c=='O' || c=='U');
}
```

This suffers from the same challenges that we presented in the introduction to switch statements: `c` gets evaluated multiple times and the reader has to make sure it is `c` that is being evaluated each time.

You can do something similar using switch statements by placing multiple case labels in sequence:

```
bool isVowel(char c)
{
    switch (c)
    {
        case 'a': // if c is 'a'
        case 'e': // or if c is 'e'
        case 'i': // or if c is 'i'
        case 'o': // or if c is 'o'
        case 'u': // or if c is 'u'
        case 'A': // or if c is 'A'
        case 'E': // or if c is 'E'
        case 'I': // or if c is 'I'
        case 'O': // or if c is 'O'
        case 'U': // or if c is 'U'
            return true;
        default:
            return false;
    }
}
```

Remember, execution begins at the first statement after a matching case label. Case labels aren't statements (they're labels), so they don't count.

The first statement after *all* of the case statements in the above program is `return true`, so if any case labels match, the function will return `true`.

Thus, we can "stack" case labels to make all of those case labels share the same set of statements afterward. This is not considered fallthrough behavior, so use of comments or `[[fallthrough]]` is not needed here.

Switch case scoping

With `if statements`, you can only have a single statement after the if-condition, and that statement is considered to be implicitly inside a block:

```
if (x > 10)
    std::cout << x << " is greater than 10\n"; // this line implicitly considered to be inside a block
```

However, with switch statements, the statements after labels are all scoped to the the switch block. No implicit blocks are created.

```
switch (1)
{
    case 1:
        foo();
        break;
    default:
        std::cout << "default case\n";
        break;
}
```

In the above example, the 2 statements between the `case 1` and the default label are scoped as part of the switch block, not a block implicit to `case 1`.

Variable declaration and initialization inside case statements

You can declare (but not initialize) variables inside the switch, both before and after the case labels:

```
switch (1)
{
    int a; // okay: declaration is allowed before the case labels
    int b{ 5 }; // illegal: initialization is not allowed before the case labels

    case 1:
        int y; // okay but bad practice: declaration is allowed within a case
        y = 4; // okay: assignment is allowed
        break;

    case 2:
        y = 5; // okay: y was declared above, so we can use it here too
        break;

    case 3:
        int z{ 4 }; // illegal: initialization is not allowed within a case
        break;
}
```

Note that although variable `y` was defined in `case 1`, it was used in `case 2` as well. Because the statements under each case are not inside an implicit block, that means all statements inside the switch are part of the same scope. Thus, a variable defined in one case can be used in a later case, even if the case in which the variable is defined is never executed! Put another way, defining a variable without an initializer is just telling the compiler that the variable is now in scope from that point on. This doesn't require the definition to actually be executed.

However, initialization of variables is disallowed and will cause a compile error. This is because initializing a variable *does* require execution, and initialization could be skipped over depending on which cases are executed.

If a case needs to define and/or initialize a new variable, best practice is to do so inside a block underneath the case statement:

```
switch (1)
{
    case 1:
    { // note addition of block here
        int x{ 4 }; // okay, variables can be initialized inside a block inside a case
        std::cout << x;
        break;
    }
    default:
        std::cout << "default case\n";
        break;
}
```

Rule

If defining variables used in a case statement, do so in a block inside the case (or before the switch if appropriate)

Quiz time

Question #1

Write a function called calculate() that takes two integers and a char representing one of the following mathematical operations: +, -, *, /, or % (modulus). Use a switch statement to perform the appropriate mathematical operation on the integers, and return the result. If an invalid

operator is passed into the function, the function should print an error. For the division operator, do an integer division.

Hint: “operator” is a keyword, variables can’t be named “operator”.

Show Solution

Next lesson
7.6 Goto statements

Back to table of contents

Previous lesson
7.4 Switch statement basics

Notify me about replies: POST COMMENT

DP N N FOUT

Newest