8.5 — Switch statement basics

Although it is possible to chain many if-else statements together, this is both difficult to read and inefficient. Consider the following program:

```
#include <iostream>

void printDigitName(int x)
{
    if (x == 1)
        std::cout << "One";
    else if (x == 2)
        std::cout << "Two";
    else if (x == 3)
        std::cout << "Three";
    else
        std::cout << "Unknown";
}

int main()
{
    printDigitName(2);
    std::cout << '\n';

    return 0;
}
```

Variable `x` in `printDigitName()` will be evaluated up to three times depending on the value passed in (which is inefficient), and the reader has to be sure that it is `x` being evaluated each time (not some other variable).

Because testing a variable or expression for equality against a set of different values is common, C++ provides an alternative conditional statement called a **switch statement** that is specialized for this purpose. Here is the same program as above using a switch:

```
#include <iostream>

void printDigitName(int x)
{
    switch (x)
    {
    case 1:
        std::cout << "One";
        return;
    case 2:
        std::cout << "Two";
        return;
    case 3:
        std::cout << "Three";
        return;
    default:
        std::cout << "Unknown";
        return;
    }
}

int main()
{
    printDigitName(2);
    std::cout << '\n';

    return 0;
}
```

The idea behind a **switch statement** is simple: an expression (sometimes called the `condition`) is evaluated to produce a value. If the expression's value is equal to the value after any of the `case labels`, the statements after the matching `case label` are executed. If no matching value can be found and a `default label` exists, the statements after the `default label` are executed instead.

Compared to the original `if statement`, the `switch statement` has the advantage of only evaluating the expression once (making it more efficient), and the `switch statement` also makes it clearer to the reader that it is the same expression being tested for equality in each case.

Best practice

Prefer `switch statements` over if-else chains when there is a choice.

Let's examine each of these concepts in more detail.

Starting a switch

We start a `switch statement` by using the `switch` keyword, followed by parentheses with the conditional expression that we would like to evaluate inside. Often the expression is just a single variable, but it can be any valid expression.

The one restriction is that the condition must evaluate to an integral type (see lesson 4.1 -- Introduction to fundamental data types if you need a reminder which fundamental types are considered integral types) or an enumerated type (covered in future lesson 13.2 -- Unscoped enumerations), or be convertible to one. Expressions that evaluate to floating point types, strings, and most other non-integral types may not be used here.

For advanced readers

Why does the switch type only allow for integral (or enumerated) types? The answer is because switch statements are designed to be highly optimized. Historically, the most common way for compilers to implement switch statements is via Jump tables -- and jump tables only work with integral values.

For those of you already familiar with arrays, a jump table works much like an array, an integral value is used as the array index to “jump” directly to a result. This can be much more efficient than doing a bunch of sequential comparisons.

Of course, compilers don’t have to implement switches using jump tables, and sometimes they don’t. There is technically no reason that C++ couldn’t relax the restriction so that other types could be used as well, they just haven’t done so yet (as of C++23).

Following the conditional expression, we declare a block. Inside the block, we use labels to define all of the values we want to test for equality. There are two kinds of labels used with switch statements, which we’ll discuss subsequently.

Case labels

The first kind of label is the **case label**, which is declared using the `case` keyword and followed by a constant expression. The constant expression must either match the type of the condition or must be convertible to that type.

If the value of the conditional expression equals the expression after a `case label`, execution begins at the first statement after that `case label` and then continues sequentially.

Here’s an example of the condition matching a `case label`:

```
#include <iostream>

void printDigitName(int x)
{
    switch (x) // x is evaluated to produce value 2
    {
    case 1:
        std::cout << "One";
        return;
    case 2: // which matches the case statement here
        std::cout << "Two"; // so execution starts here
        return; // and then we return to the caller
    case 3:
        std::cout << "Three";
        return;
    default:
        std::cout << "Unknown";
        return;
    }
}

int main()
{
    printDigitName(2);
    std::cout << '\n';

    return 0;
}
```

This code prints:

```
Two
```

In the above program, `x` is evaluated to produce value `2`. Because there is a case label with value `2`, execution jumps to the statement underneath that matching case label. The program prints `Two`, and then the `return statement` is executed, which returns back to the caller.

There is no practical limit to the number of case labels you can have, but all case labels in a switch must be unique. That is, you can not do this:

```
switch (x)
{
case 54:
case 54:  // error: already used value 54!
case '6': // error: '6' converts to integer value 54, which is already used
}
```

If the conditional expression does not match any of the case labels, no cases are executed. We'll show an example of this shortly.

The default label

The second kind of label is the **default label** (often called the **default case**), which is declared using the `default` keyword. If the conditional expression does not match any case label and a default label exists, execution begins at the first statement after the default label.

Here’s an example of the condition matching the default label:

```
#include <iostream>

void printDigitName(int x)
{
    switch (x) // x is evaluated to produce value 5
    {
    case 1:
        std::cout << "One";
        return;
    case 2:
        std::cout << "Two";
        return;
    case 3:
        std::cout << "Three";
        return;
    default: // which does not match to any case labels
        std::cout << "Unknown"; // so execution starts here
        return; // and then we return to the caller
    }
}

int main()
{
    printDigitName(5);
    std::cout << '\n';

    return 0;
}
```

This code prints:

```
Unknown
```

The default label is optional, and there can only be one default label per switch statement. By convention, the `default case` is placed last in the switch block.

Best practice

Place the default case last in the switch block.

No matching case label and no default case

If the value of the conditional expression does not match any of the case labels, and no default case has been provided, then no cases inside the switch are executed. Execution continues after the end of the switch block.

```
#include <iostream>

void printDigitName(int x)
{
    switch (x) // x is evaluated to produce value 5
    {
    case 1:
        std::cout << "One";
        return;
    case 2:
        std::cout << "Two";
        return;
    case 3:
        std::cout << "Three";
        return;
    // no matching case exists and there is no default case
    }

    // so execution continues here
    std::cout << "Hello";
}

int main()
{
    printDigitName(5);
    std::cout << '\n';

    return 0;
}
```

In the above example, `x` evalutes to `5`, but there is no case label matching `5`, nor is there a default case. As a result, no cases execute. Execution continues after the switch block, printing `Hello`.

Taking a break

In the above examples, we used `return statements` to stop execution of the statements after our labels. However, this also exits the entire function.

A **break statement** (declared using the `break` keyword) tells the compiler that we are done executing statements within the switch, and that execution should continue with the statement after the end of the switch block. This allows us to exit a `switch statement` without exiting the entire function.

Here's a slightly modified example rewritten using `break` instead of `return`:

```
#include <iostream>

void printDigitName(int x)
{
    switch (x) // x evaluates to 3
    {
    case 1:
        std::cout << "One";
        break;
    case 2:
        std::cout << "Two";
        break;
    case 3:
        std::cout << "Three"; // execution starts here
        break; // jump to the end of the switch block
    default:
        std::cout << "Unknown";
        break;
    }

    // execution continues here
    std::cout << " Ah-Ah-Ah!";
}

int main()
{
    printDigitName(3);
    std::cout << '\n';

    return 0;
}
```

The above example prints:

```
Three Ah-Ah-Ah!
```

Best practice

Each set of statements underneath a label should end in a `break statement` or a `return statement`. This includes the statements underneath the last label in the switch.

So what happens if you don't end a set of statements under a label with a `break` or `return`? We'll explore that topic, and others, in the next lesson.

Labels are conventionally not indented

In lesson 2.9 -- Naming collisions and an introduction to namespaces, we noted that code is typically indented one level to help identify that it's part of a nested scope region. Since the curly braces of the switch define a new scope region, we're normally indent everything inside the curly braces one level.

A label, on the other hand, does not define a nested scope. Therefore, the code following a label is conventionally not indented.

However, if we indent both the labels and the subsequent statements to the same level, we end up with something like this:

```
// Unreadable version
void printDigitName(int x)
{
    switch (x)
    {
        case 1:
        std::cout << "One";
        return;
        case 2:
        std::cout << "Two";
        return;
        case 3:
        std::cout << "Three";
        return;
        default:
        std::cout << "Unknown";
        return;
    }
}
```

This makes it really hard to determine where each case starts and ends.

We have two choices here. First, we can indent the statements following the labels anyway:

```
// Acceptable but not preferred version
void printDigitName(int x)
{
    switch (x)
    {
        case 1: // indented from switch block
            std::cout << "One"; // indented from label (misleading)
            return;
        case 2:
            std::cout << "Two";
            return;
        case 3:
            std::cout << "Three";
            return;
        default:
            std::cout << "Unknown";
            return;
    }
}
```

While this is certainly more readable than the prior version, it implies that the statements beneath each label are in a nested scope, which is not the case (we’ll see examples of this in the next lesson, where a variable we define in one case can be used in another case). This formatting is considered acceptable (as it is readable), but it is not preferred.

Conventionally, labels are simply not indented:

```
// Preferred version
void printDigitName(int x)
{
    switch (x)
    {
    case 1: // not indented from switch statement
        std::cout << "One";
        return;
    case 2:
        std::cout << "Two";
        return;
    case 3:
        std::cout << "Three";
        return;
    default:
        std::cout << "Unknown";
        return;
    }
}
```

This makes it easy to identify each label. And because the statements are only indented one level from the switch block, it correctly implies that the statements are all part of the scope of the switch block.

In future lessons, we’ll encounter other types of labels -- these are also conventionally not indented for the same reason.

Best practice

Prefer not to indent labels. This allows them to stand out from the surrounding code without implying that they are defining a nested scope region.