5.3 — Switch statements

BY ALEX ON JUNE 21ST, 2007 | LAST MODIFIED BY ALEX ON JUNE 7TH, 2018

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

```
#include <iostream>

enum Colors
{
    COLOR_BLACK,
    COLOR_WHITE,
    COLOR_RED,
    COLOR_GREEN,
    COLOR_BLUE
};

void printColor(Colors color)
{
    if (color == COLOR_BLACK)
        std::cout << "Black";
    else if (color == COLOR_WHITE)
        std::cout << "White";
    else if (color == COLOR_RED)
        std::cout << "Red";
    else if (color == COLOR_GREEN)
        std::cout << "Green";
    else if (color == COLOR_BLUE)
        std::cout << "Blue";
    else
        std::cout << "Unknown";
}

int main()
{
    printColor(COLOR_GREEN);

    return 0;
}
```

Because doing if-else chains on a single variable testing for equality is so common, C++ provides an alternative conditional branching operator called a **switch**. Here is the same program as above in switch form:

```
void printColor(Colors color)
{
    switch (color)
    {
        case COLOR_BLACK:
            std::cout << "Black";
            break;
        case COLOR_WHITE:
            std::cout << "White";
            break;
        case COLOR_RED:
            std::cout << "Red";
            break;
        case COLOR_GREEN:
            std::cout << "Green";
            break;
        case COLOR_BLUE:
            std::cout << "Blue";
            break;
        default:
            std::cout << "Unknown";
```

```
            break;
    }
}
```

The overall idea behind switch statements is simple: the switch expression is evaluated to produce a value, and each case label is tested against this value for equality. If a case label matches, the statements after the case label are executed. If no case label matches the switch expression, the statements after the default label are executed (if it exists).

Because of the way they are implemented, switch statements are typically more efficient than if-else chains.

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 the expression that we would like to evaluate. Typically this expression is just a single variable, but it can be something more complex like `nX + 2` or `nX - nY`. The one restriction on this expression is that it must evaluate to an integral type (that is, char, short, int, long, long long, or enum). Floating point variables and other non-integral types may not be used here.

Following the switch 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.

Case labels

The first kind of label is the **case label**, which is declared using the **case** keyword and followed by a constant expression. A constant expression is one that evaluates to a constant value -- in other words, either a literal (such as 5), an enum (such as COLOR_RED), or a constant variable (such as x, when x has been defined as a const int).

The constant expression following the case label is tested for equality against the expression following the switch keyword. If they match, the code under the case label is executed.

It is worth noting that all case label expressions must evaluate to a unique value. That is, you can not do this:

```
switch (x)
{
    case 4:
    case 4:  // illegal -- already used value 4!
    case COLOR_BLUE: // illegal, COLOR_BLUE evaluates to 4!
};
```

It is possible to have multiple case labels refer to the same statements. The following function uses multiple cases to test if the 'c' parameter is an ASCII digit.

```
bool isDigit(char c)
{
    switch (c)
    {
        case '0': // if c is 0
        case '1': // or if c is 1
        case '2': // or if c is 2
        case '3': // or if c is 3
        case '4': // or if c is 4
        case '5': // or if c is 5
        case '6': // or if c is 6
        case '7': // or if c is 7
        case '8': // or if c is 8
        case '9': // or if c is 9
            return true; // then return true
        default:
            return false;
    }
}
```

In the case where c is an ASCII digit, the first statement after the matching case statement is executed, which is "return true".

The default label

The second kind of label is the **default label** (often called the “default case”), which is declared using the **default** keyword. The code under this label gets executed if none of the cases match the switch expression. The default label is optional, and there can only be one default label per switch statement. It is also typically declared as the last label in the switch block, though this is not strictly necessary.

In the isDigit() example above, if c is not an ASCII digit, the default case executes and returns false.

Switch execution and fall-through

One of the trickiest things about case statements is the way in which execution proceeds when a case is matched. When a case is matched (or the default is executed), execution begins at the first statement following that label and continues until one of the following termination conditions is true:
1) The end of the switch block is reached
2) A return statement occurs
3) A goto statement occurs
4) A break statement occurs
5) Something else interrupts the normal flow of the program (e.g. a call to exit(), an exception occurs, the universe implodes, etc…)

Note that if none of these termination conditions are met, cases will overflow into subsequent cases! Consider the following snippet:

```
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
}
```

This snippet prints the result:

```
2
3
4
5
```

This is probably not what we wanted! When execution flows from one case into another case, this is called **fall-through**. Fall-through is almost never desired by the programmer, so in the rare case where it is, it is common practice to leave a comment stating that the fall-through is intentional.

Break statements

A **break statement** (declared using the **break** keyword) tells the compiler that we are done with this switch (or while, do while, or for loop). After a break statement is encountered, execution continues with the statement after the end of the switch block.

Let’s look at our last example with break statements properly inserted:

```
switch (2)
{
   case 1: // Does not match -- skipped
       std::cout << 1 << '\n';
       break;
   case 2: // Match!  Execution begins at the next statement
       std::cout << 2 << '\n'; // Execution begins here
       break; // Break terminates the switch statement
   case 3:
       std::cout << 3 << '\n';
       break;
   case 4:
```

```
        std::cout << 4 << '\n';
        break;
    default:
        std::cout << 5 << '\n';
        break;
}
// Execution resumes here
```

Now, when case 2 matches, the integer 2 will be output, and the break statement will cause the switch to terminate. The other cases are skipped.

Warning: Forgetting the break statement at the end of the case statements is one of the most common C++ mistakes made!

Multiple statements inside a switch block

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, you are allowed to have multiple statements after a case label, and they are not considered to be inside an implicit block.

```
switch (1)
{
    case 1:
        std::cout << 1;
        foo();
        std::cout << 2;
        break;
    default:
        std::cout << "default case\n";
        break;
}
```

In the above example, the 4 statements between the case label and default label are part of case 1, but not considered to be inside an implicit block.

If this seems a bit inconsistent, it is.

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, declaration is allowed within a case
        y = 4; // okay, this is an assignment
        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;

    default:
        std::cout << "default case" << std::endl;
        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 another case, even if the case in which the variable is defined is never executed!

This may seem a bit counter-intuitive, so let's examine why. When you define a local variable like "int y;", the variable isn't created at that point -- it's actually created at the start of the block it's declared in. However, it is not visible (in scope) until the point of declaration. The declaration statement doesn't need to execute -- it just tells the compiler that the variable can be used past that point. So with that in mind, it's a little less weird that a variable declared in one case statement can be used in another cases statement, even if the case statement that declares the variable is never executed.

However, initialization of variables directly underneath a case label is disallowed and will cause a compile error. This is because initializing a variable *does* require execution, and the case statement containing the initialization may not be 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" << std::endl;
        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

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.

2) Define an enum (or enum class, if using a C++11 capable compiler) named Animal that contains the following animals: pig, chicken, goat, cat, dog, ostrich. Write a function named getAnimalName() that takes an Animal parameter and uses a switch statement to return the name for that animal as a std::string. Write another function named printNumberOfLegs() that uses a switch statement to print the number of legs each animal walks on. Make sure both functions have a default case that prints an error message. Call printNumberOfLegs() from main() with a cat and a chicken. Your output should look like this:

```
A cat has 4 legs.
A chicken has 2 legs.
```

Quiz answers

1) **Hide Solution**

```
#include <iostream>

int calculate(int x, int y, char op)
{
    switch (op)
    {
        case '+':
            return x + y;
        case '-':
            return x - y;
        case '*':
            return x * y;
        case '/':
            return x / y;
        case '%':
            return x % y;
```

```
        default:
            std::cout << "calculate(): Unhandled case\n";
            return 0;
    }
}

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

    std::cout << "Enter another integer: ";
    int y;
    std::cin >> y;

    std::cout << "Enter a mathematical operator (+, -, *, /, or %): ";
    char op;
    std::cin >> op;

    std::cout << x << " " << op << " " << y << " is " << calculate(x, y, op) << "\n";

    return 0;
}
```

2) **Hide Solution**

```
// Note: this solution is the non-C++11 solution using a regular enum, not an enum class
#include <iostream>
#include <string>

enum Animal
{
    ANIMAL_PIG,
    ANIMAL_CHICKEN,
    ANIMAL_GOAT,
    ANIMAL_CAT,
    ANIMAL_DOG,
    ANIMAL_OSTRICH
};

std::string getAnimalName(Animal animal)
{
    switch (animal)
    {
        case ANIMAL_CHICKEN:
            return "chicken";
        case ANIMAL_OSTRICH:
            return "ostrich";
        case ANIMAL_PIG:
            return "pig";
        case ANIMAL_GOAT:
            return "goat";
        case ANIMAL_CAT:
            return "cat";
        case ANIMAL_DOG:
            return "dog";

        default:
            return "???";
    }
}

void printNumberOfLegs(Animal animal)
{
    std::cout << "A " << getAnimalName(animal) << " has ";

```

```
    switch (animal)
    {
        case ANIMAL_CHICKEN:
        case ANIMAL_OSTRICH:
            std::cout << "2";
            break;

        case ANIMAL_PIG:
        case ANIMAL_GOAT:
        case ANIMAL_CAT:
        case ANIMAL_DOG:
            std::cout << "4";
            break;

        default:
            std::cout << "???";
            break;
    }

    std::cout << " legs.\n";
}

int main()
{
    printNumberOfLegs(ANIMAL_CAT);
    printNumberOfLegs(ANIMAL_CHICKEN);

    return 0;
}
```


5.4 -- Goto statements


Index


5.2 -- If statements

Share this:

Facebook Twitter Google Pinterest

C++ TUTORIAL | PRINT THIS POST

238 comments to 5.3 — Switch statements

« Older Comments 1 2 3 4

Tamara
July 17, 2018 at 9:39 am · Reply

Hi, I'm having trouble understanding as to why it's illegal to initialize a variable directly underneath a case statement, but it's legal to initialize it if it's inside a block?

Idda