8.11 — Break and continue

Break

Although you have already seen the `break` statement in the context of `switch` statements (8.5 -- Switch statement basics), it deserves a fuller treatment since it can be used with other types of control flow statements as well. The **break statement** causes a while loop, do-while loop, for loop, or switch statement to end, with execution continuing with the next statement after the loop or switch being broken out of.

Breaking a switch

In the context of a `switch` statement, a `break` is typically used at the end of each case to signify the case is finished (which prevents fallthrough into subsequent cases):

```
#include <iostream>

void printMath(int x, int y, char ch)
{
    switch (ch)
    {
    case '+':
        std::cout << x << " + " << y << " = " << x + y << '\n';
        break; // don't fall-through to next case
    case '-':
        std::cout << x << " - " << y << " = " << x - y << '\n';
        break; // don't fall-through to next case
    case '*':
        std::cout << x << " * " << y << " = " << x * y << '\n';
        break; // don't fall-through to next case
    case '/':
        std::cout << x << " / " << y << " = " << x / y << '\n';
        break;
    }
}

int main()
{
    printMath(2, 3, '+');

    return 0;
}
```

See lesson 8.6 -- Switch fallthrough and scoping for more information about fallthrough, along with some additional examples.

Breaking a loop

In the context of a loop, a `break` statement can be used to end the loop early. Execution continues with the next statement after the end of the loop.

For example:

```
#include <iostream>

int main()
{
    int sum{ 0 };

    // Allow the user to enter up to 10 numbers
    for (int count{ 0 }; count < 10; ++count)
    {
        std::cout << "Enter a number to add, or 0 to exit: ";
        int num{};
        std::cin >> num;

        // exit loop if user enters 0
        if (num == 0)
            break; // exit the loop now

        // otherwise add number to our sum
        sum += num;
    }

    // execution will continue here after the break
    std::cout << "The sum of all the numbers you entered is: " << sum << '\n';

    return 0;
}
```

This program allows the user to type up to 10 numbers, and displays the sum of all the numbers entered at the end. If the user enters 0, the break causes the loop to terminate early (before 10 numbers have been entered).

Here's a sample execution of the above program:

```
Enter a number to add, or 0 to exit: 5
Enter a number to add, or 0 to exit: 2
Enter a number to add, or 0 to exit: 1
Enter a number to add, or 0 to exit: 0
The sum of all the numbers you entered is: 8
```

A `break` is also a common way to get out of an intentional infinite loop:

```
#include <iostream>

int main()
{
    while (true) // infinite loop
    {
        std::cout << "Enter 0 to exit or any other integer to continue: ";
        int num{};
        std::cin >> num;

        // exit loop if user enters 0
        if (num == 0)
            break;
    }

    std::cout << "We're out!\n";

    return 0;
}
```

A sample run of the above program:

```
Enter 0 to exit or any other integer to continue: 5
Enter 0 to exit or any other integer to continue: 3
Enter 0 to exit or any other integer to continue: 0
We're out!
```

Break vs return

New programmers sometimes have trouble understanding the difference between `break` and `return`. A `break` statement terminates the switch or loop, and execution continues at the first statement beyond the switch or loop. A `return` statement terminates the entire function that the loop is within, and execution continues at point where the function was called.

```
#include <iostream>

int breakOrReturn()
{
    while (true) // infinite loop
    {
        std::cout << "Enter 'b' to break or 'r' to return: ";
        char ch{};
        std::cin >> ch;

        if (ch == 'b')
            break; // execution will continue at the first statement beyond the loop

        if (ch == 'r')
            return 1; // return will cause the function to immediately return to the
caller (in this case, main())
    }

    // breaking the loop causes execution to resume here

    std::cout << "We broke out of the loop\n";

    return 0;
}

int main()
{
    int returnValue{ breakOrReturn() };
    std::cout << "Function breakOrReturn returned " << returnValue << '\n';

    return 0;
}
```

Here are two runs of this program:

```
Enter 'b' to break or 'r' to return: r
Function breakOrReturn returned 1

Enter 'b' to break or 'r' to return: b
We broke out of the loop
Function breakOrReturn returned 0
```

Continue

The **continue statement** provides a convenient way to end the current iteration of a loop without terminating the entire loop.

Here’s an example of using continue:

```
#include <iostream>

int main()
{
    for (int count{ 0 }; count < 10; ++count)
    {
        // if the number is divisible by 4, skip this iteration
        if ((count % 4) == 0)
            continue; // go to next iteration

        // If the number is not divisible by 4, keep going
        std::cout << count << '\n';

        // The continue statement jumps to here
    }

    return 0;
}
```

This program prints all of the numbers from 0 to 9 that aren't divisible by 4:

```
1
2
3
5
6
7
9
```

A `continue` statement works by causing the current point of execution to jump to the bottom of the current loop.

In the case of a for-loop, the end-statement of the for-loop (in the above example, `++count`) still executes after a continue (since this happens after the end of the loop body).

Be careful when using a `continue` statement with while or do-while loops. These loops typically change the value of variables used in the condition inside the loop body. If use of a `continue` statement causes these lines to be skipped, then the loop can become infinite!

Consider the following program:

```
#include <iostream>

int main()
{
    int count{ 0 };
    while (count < 10)
    {
        if (count == 5)
            continue; // jump to end of loop body

        std::cout << count << '\n';

        ++count; // this statement is never executed after count reaches 5

        // The continue statement jumps to here
    }

    return 0;
}
```

This program is intended to print every number between 0 and 9 except 5. But it actually prints:

```
0
1
2
3
4
```

and then goes into an infinite loop. When `count` is `5`, the `if` statement evaluates to `true`, and the `continue` causes the execution to jump to the bottom of the loop. The `count` variable is never incremented. Consequently, on the next pass, `count` is still `5`, the `if` statement is still `true`, and the program continues to loop forever.

Of course, you already know that if you have an obvious counter variable, you should be using a `for` loop, not a `while` or `do while` loop.

The debate over use of break and continue

Many textbooks caution readers not to use `break` and `continue` in loops, both because it causes the execution flow to jump around, and because it can make the flow of logic harder to follow. For example, a `break` in the middle of a complicated piece of logic could either be missed, or it may not be obvious under what conditions it should be triggered.

However, used judiciously, `break` and `continue` can help make loops more readable by keeping the number of nested blocks down and reducing the need for complicated looping logic.

For example, consider the following program:

```
#include <iostream>

int main()
{
    int count{ 0 }; // count how many times the loop iterates
    bool keepLooping { true }; // controls whether the loop ends or not
    while (keepLooping)
    {
        std::cout << "Enter 'e' to exit this loop or any other character to continue:
";
        char ch{};
        std::cin >> ch;

        if (ch == 'e')
            keepLooping = false;
        else
        {
            ++count;
            std::cout << "We've iterated " << count << " times\n";
        }
    }

    return 0;
}
```

This program uses a Boolean variable to control whether the loop continues or not, as well as a nested block that only runs if the user doesn't exit.

Here's a version that's easier to understand, using a `break` statement:

```
#include <iostream>

int main()
{
    int count{ 0 }; // count how many times the loop iterates
    while (true) // loop until user terminates
    {
        std::cout << "Enter 'e' to exit this loop or any other character to continue:
";
        char ch{};
        std::cin >> ch;

        if (ch == 'e')
            break;

        ++count;
        std::cout << "We've iterated " << count << " times\n";
    }

    return 0;
}
```

In this version, by using a single `break` statement, we've avoided the use of a Boolean variable (and having to understand both what its intended use is, and where its value is changed), an `else` statement, and a nested block.

The `continue` statement is most effectively used at the top of a for-loop to skip loop iterations when some condition is met. This can allow us to avoid a nested block.

For example, instead of this:

```
#include <iostream>

int main()
{
    for (int count{ 0 }; count < 10; ++count)
    {
        // if the number is not divisible by 4...
        if ((count % 4) != 0) // nested block
        {
                // Print the number
                std::cout << count << '\n';
        }
    }

    return 0;
}
```

We can write this:

```
#include <iostream>

int main()
{
    for (int count{ 0 }; count < 10; ++count)
    {
        // if the number is divisible by 4, skip this iteration
        if ((count % 4) == 0)
            continue;

        // no nested block needed

        std::cout << count << '\n';
    }

    return 0;
}
```

Minimizing the number of variables used and keeping the number of nested blocks down both improve code comprehensibility more than a `break` or `continue` harms it. For that reason, we believe judicious use of `break` or `continue` is acceptable.

Best practice

Use break and continue when they simplify your loop logic.

The debate over use of early returns

There's a similar argument to be made for return statements. A return statement that is not the last statement in a function is called an **early return**. Many programmers believe early returns should be avoided. A function that only has one return statement at the bottom of the function has a simplicity to it -- you can assume the function will take its arguments, do whatever logic it has implemented, and return a result without deviation. Having extra returns complicates the logic.

The counter-argument is that using early returns allows your function to exit as soon as it is done, which reduces having to read through unnecessary logic and minimizes the need for conditional nested blocks, which makes your code more readable.

Some developers take a middle ground, and only use early returns at the top of a function to do parameter validation (catch bad arguments passed in), and then a single return thereafter.

Our stance is that early returns are more helpful than harmful, but we recognize that there is a bit of art to the practice.

Best practice

Use early returns when they simplify your function's logic.