5.8 — Break and continue

BY ALEX ON JUNE 26TH, 2007 | LAST MODIFIED BY ALEX ON JUNE 29TH, 2018

Break

Although you have already seen the break statement in the context of switch statements, it deserves a fuller treatment since it can be used with other types of loops as well. The break statement causes a do, for, switch, or while statement to terminate.

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 fall-through):

```
switch (ch)
{
    case '+':
        doAddition(x, y);
        break;
    case '-':
        doSubtraction(x, y);
        break;
    case '*':
        doMultiplication(x, y);
        break;
    case '/':
        doDivision(x, y);
        break;
}
```

Breaking a loop

In the context of a loop, a break statement can be used to cause the loop to terminate early:

```
#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;

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

    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).

Note that break can be used to get out of an infinite loop:

```cpp
#include <iostream>

int main()
{
    while (true) // infinite loop
    {
        std::cout << "Enter 0 to exit or anything else 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;
}
```

Break vs return

New programmers often 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.

```cpp
#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 ca
se, 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 breakOrContinue returned " << returnValue << '\n';

    return 0;
}
```

Continue

The continue statement provides a convenient way to jump to the end of the loop body for the current iteration. This is useful when we want to terminate the current iteration early.

Here's an example of using continue:

```cpp
for (int count=0; count  < 20; ++count)
```

```
{
    // if the number is divisible by 4, skip this iteration
    if ((count % 4) == 0)
        continue; // jump to end of loop body

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

    // The continue statement jumps to here
}
```

This program prints all of the numbers from 0 to 19 that aren't divisible by 4.

In the case of a for loop, the end-statement of the for loop 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. Because these loops typically increment the loop variables in the body of the loop, using continue can cause the loop to become infinite! Consider the following program:

```
int count(0);
while (count < 10)
{
    if (count == 5)
        continue; // jump to end of loop body
    std::cout << count << " ";
    ++count;

    // The continue statement jumps to here
}
```

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 loop jumps to the bottom. 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.

Here's an example with a do-while loop using continue correctly:

```
int count(0);
do
{
    if (count == 5)
        continue; // jump to end of loop body
    std::cout << count << " ";

    // The continue statement jumps to here
} while (++count < 10); // this still executes since it's outside the loop body
```

This prints:

```
0 1 2 3 4 6 7 8 9
```

Using break and continue

Many textbooks caution readers not to use break and continue, 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 it is set), an else statement, and a nested block.

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


5.9 -- Random number generation


Index