8.9 — Do while statements

Consider the case where we want to show the user a menu and ask them to make a selection -- and if the user chooses an invalid selection, to ask them again. Clearly the menu and selection should go inside a loop of some kind (so we can keep asking the user until they enter valid input), but what kind of loop should we choose?

Since a while loop evaluates the condition up front, it's an awkward choice. We could solve the issue like this:

```
#include <iostream>

int main()
{
    // selection must be declared outside while-loop, so we can use it later
    int selection{ 0 };

    while (selection != 1 && selection != 2 &&
        selection != 3 && selection != 4)
    {
        std::cout << "Please make a selection: \n";
        std::cout << "1) Addition\n";
        std::cout << "2) Subtraction\n";
        std::cout << "3) Multiplication\n";
        std::cout << "4) Division\n";
        std::cin >> selection;
    }

    // do something with selection here
    // such as a switch statement

    std::cout << "You selected option #" << selection << '\n';

    return 0;
}
```

But this only works because our initial value of `0` for `selection` isn't in the set of valid values (`1, 2, 3 or 4`). What if `0` was a valid choice? We'd have to pick a different initializer to represent "invalid" -- and now we're introducing magic numbers (5.2 -- Literals) into our code.

We could instead add a new variable to track validity, like so:

```
#include <iostream>

int main()
{
    int selection { 0 };
    bool invalid { true }; // new variable just to gate the loop

    while (invalid)
    {
        std::cout << "Please make a selection: \n";
        std::cout << "1) Addition\n";
        std::cout << "2) Subtraction\n";
        std::cout << "3) Multiplication\n";
        std::cout << "4) Division\n";

        std::cin >> selection;
        invalid = (selection < 1 || selection > 4);
    }

    // do something with selection here
    // such as a switch statement

    std::cout << "You selected option #" << selection << '\n';

    return 0;
}
```

While this avoids the magic number, it introduces a new variable just to ensure the loop runs once, and that adds complexity and the possibility of additional errors.

Do while statements

To help solve problems like the above, C++ offers the do-while statement:

```
do
    statement; // can be a single statement or a compound statement
while (condition);
```

A **do while statement** is a looping construct that works just like a while loop, except the statement always executes at least once. After the statement has been executed, the do-while loop checks the condition. If the condition evaluates to `true`, the path of execution jumps back to the top of the do-while loop and executes it again.

Here is our example above using a do-while loop instead of a while loop:

```
#include <iostream>

int main()
{
    // selection must be declared outside of the do-while-loop, so we can use it
later
    int selection{};

    do
    {
        std::cout << "Please make a selection: \n";
        std::cout << "1) Addition\n";
        std::cout << "2) Subtraction\n";
        std::cout << "3) Multiplication\n";
        std::cout << "4) Division\n";
        std::cin >> selection;
    }
    while (selection < 1 || selection > 4);

    // do something with selection here
    // such as a switch statement

    std::cout << "You selected option #" << selection << '\n';

    return 0;
}
```

In this way, we've avoided both magic numbers and additional variables.

One thing worth discussing in the above example is that the `selection` variable must be declared outside of the do block. If the `selection` variable were to be declared inside the do block, it would be destroyed when the do block terminates, which happens before the conditional is evaluated. But we need the variable in the while conditional -- consequently, the `selection` variable must be declared outside the do block (even if it wasn't used later in the body of the function).

In practice, do-while loops aren't commonly used. Having the condition at the bottom of the loop obscures the loop condition, which can lead to errors. Many developers recommend avoiding do-while loops altogether as a result. We'll take a softer stance and advocate for preferring while loops over do-while when given an equal choice.

Best practice

Favor while loops over do-while when given an equal choice.