7.6 — Goto statements

ALEX AUGUST 6, 2021

The next kind of control flow statement we'll cover is the unconditional jump. An unconditional jump causes execution to jump to another spot in the code. The term "unconditional" means the jump always happens (unlike an `if statement` or `switch statement`, where the jump only happens conditionally based on the result of an expression).

In C++, unconditional jumps are implemented via agoto statement, and the spot to jump to is identified through use of a statement label. The following is an example of a `goto statement` and `statement label`:

```
#include <iostream>
#include <cmath> // for sqrt() function

int main()
{
    double x{};
tryAgain: // this is a statement label
    std::cout << "Enter a non-negative number: ";
    std::cin >> x;

    if (x < 0.0)
        goto tryAgain; // this is the goto statement

    std::cout << "The square root of " << x << " is " << std::sqrt(x) <<
'\n';
    return 0;
}
```

In this program, the user is asked to enter a non-negative number. However, if a negative number is entered, the program utilizes a `goto statement` to jump back to the tryAgain label. The user is then asked again to enter a new number. In this way, we can continually ask the user for input until he or she enters something valid.

Here's a sample run of this program:

```
Enter a non-negative number: -4
Enter a non-negative number: 4
The square root of 4 is 2
```

Statement labels have function scope

In the chapter on object scope (chapter 6), we covered two kinds of scope: local (block) scope, and file (global) scope. Statement labels utilize a third kind of scope: function scope, which means the label is visible throughout the function even before its point of declaration. The `goto statement` and its corresponding `statement label` must appear in the same function.

While the above example shows a `goto statement` that jumps backwards (to a preceding point in the function), `goto statements` can also jump forward:

```
#include <iostream>

void printCats(bool skip)
{
    if (skip)
        goto end; // jump forward; statement label 'end' is visible here due to it having function
scope

    std::cout << "cats";
end:
    ; // statement labels must be associated with a statement
}

int main()
{
    printCats(true);  // jumps over the print statement and doesn't print anything
    printCats(false); // prints "cats"

    return 0;
}
```

This prints:

```
cats
```

Beyond the jumping forward, there are a couple of interesting things worth mentioning in the program above.

First, note that `statement labels` must be associated with a statement (hence their name: they label a statement). Because the end of the function had no statement, we had to use a `null statement` so we had a statement to label. Second, we were able to jump to the statement labeled by `end` even though we hadn't declared `end` yet due to `statement labels` having function scope. No forward declaration of `statement labels` is necessary. Third, it's worth explicitly mentioning that the above program is poor form -- it would have been better to use an `if statement` to skip the print statement than a `goto statement` to jump over it.

There are two primary limitations to jumping: You can jump only within the bounds of a single function (you can't jump out of one function and into another), and if you jump forward, you can't jump forward over the initialization of any variable that is still in scope at the location being jumped to. For example:

```
int main()
{
    goto skip;   // error: this jump is illegal because...
    int x { 5 }; // this initialized variable is still in scope at statement label
'skip'
skip:
    x += 3;      // what would this even evaluate to if x wasn't initialized?
    return 0;
}
```

Note that you can jump backwards over a variable initialization, and the variable will be re-initialized when the initialization is executed.

Avoid using goto

Use of `goto` is shunned in C++ (and other modern high level languages as well). Edsger W. Dijkstra, a noted computer scientist, laid out the case for avoiding `goto` in a famous but difficult to read paper called Go To Statement Considered Harmful. The primary problem with `goto` is that it allows a programmer to jump around the code arbitrarily. This creates what is not-so-affectionately known as `spaghetti code`. Spaghetti code is code that has a path of execution that resembles a bowl of spaghetti (all tangled and twisted), making it extremely difficult to follow the logic of such code.

As Dijkstra says somewhat humorously, "the quality of programmers is a decreasing function of the density of go to statements in the programs they produce".

Almost any code written using a `goto statement` can be more clearly written using other constructs in C++, such as `if statements` and loops. One notable exception is when you need to exit a nested loop but not the entire function -- in such a case, a `goto` to just beyond the loops is probably the cleanest solution.

Best practice

Avoid `goto statements` (unless the alternatives are significantly worse for code readability).

Next lesson
7.7 Intro to loops and while statements

Back to table of contents

Previous lesson
7.5 Switch fallthrough and scoping

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

DP N N FOUT

Newest