Labels, gotos, switches, and breaks

Ben Klemens

14 August 2011

Within the C syntax, there is a subset that (1) is delightfully internally consistent and elegant and (2) entirely unnecessary. For the sake of saving space in the book and the reader's mind, I cut it this segment from *Modeling with Data*. But I'm reprinting it here because this is entirely valid syntax, and if you're advanced enough that you're reading other people's code, you'll need to be familiar with these things. I like C for being an example of how far you can get with just a few keywords, but I like this subset of C for being a self-contained, optional, and occasionally useful addition.

[I've been convinced to switch to a more script-like indentation format, by the way, wherein one-line `if` statements are actually put on one line, and those with curly braces get several lines.]

The purpose of all of these elements is to jump to a different place in the code listing. There was once a time when this was common, because it was basically all a programmer had on hand to work with, but in the present day the `goto` is considered harmful[1] (PDF). If there are multiple jumps to be followed, a reader can easily get lost.

However, a single jump by itself tends to be relatively easy to follow, and can clarify if used appropriately and in moderation. Linus Torvalds, the author of the Linux kernel, recommends the `goto`[2] for limited uses like cutting out of a function when there's an error or processing is done, as in the example below.

Labels and `break` A line of code can be named, by simply providing a name with a colon after it. You can then jump to that line via `goto`. Here is a code snippet that presents the basic idea, with a line labeled `outro`. It finds the sum of all the elements in two vectors, provided they are all not `NaN`. If one of the elements is `NaN`, this is an error and we need to exit the function. But however we choose to exit, we will free both vectors as cleanup. We could place the cleanup code in the listing three times (once if `vector` has a `NaN`, once if `vector2` has one, and once on OK exit), but it's cleaner to have one exit segment and jump to it as needed.

```
double sum=0, ok=0;
for (int i=0; i< vector_size; i++){
    if (isnan (vector[i])) goto outro;
    sum += vector[i];
}
```

[1] `https://www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF`

[2] `http://kerneltrap.org/node/553/2131`

```
for (int i=0; i< vector2_size; i++){
    if (isnan (vector2[i])) goto outro;
    sum += vector2[i];
}
ok=1;

outro:
printf("The sum until the first NaN (if any) was %g\n", sum);
free(vector);
free(vector2);
return ok;
```

The `goto` will only work within one function. If you need to jump from one function to an entirely different one, check your standard library documentation (or the man pages) for `longjmp`.

An alternative is `break`, which cuts out of the innermost loop in which it is located. It is primarily useful when searching through an array for an item. Once you have found what you are looking for, there is no need to continue looping to the end of the array. Here we find the first element greater than a cutoff. If it isn't found, return `-INFINITY`. Here is code that would work about like the above:

```
double out = -INFINITY
for (int i=0; i< vector_size; i++){
    if (a_vector[i] > cutoff){
        out = a_vector[i];
        break;
    }
}
free(a_vector);
free(another_vector);
return out;
```

The `continue` keyword is a natural complement to `break`, telling the computer to break from just this iteration of the loop, but continue stepping through the loop. Say that we have decided that it is OK to ignore `NaN`s in our data. Then the following example will sum the non-`NaN` elements of the vector. If we find that element `i` is `NaN`, then we continue back to the top of the loop without doing anything with that element.

```
double sum_of_vals=0;
for (int i=0; i< vector_size; i++){
    if (isnan(a_vector[i])) continue;
    sum_of_vals += a_vector[i];
}
return sum_of_vals;
```

Switch The `switch` statement provides a way to branch among many options. First, you will need a variable indicating a single character. For example, Listing 6.13 of *Modeling with Data* (p 209) uses the `getopt` function to read command line arguments. That function will return a single character, and then a `switch` statement can branch depending on the value returned.

```
char c;
while ((c = get_opt(...))){ //See Section 6.3 (p 203) for details.
    switch(c){
       case 'v':
            verbose++;
            break;
       case 'w':
            weighting_function();
            break;
       case 'f':
            fun_function();
            break;
    }
}
```

So when `c == 'v'`, the verbosity level is increased, when `c == 'w'`, the weighting function is called, et cetera.

Note well the abundance of `break` statements (which cut to the end of the `switch` statement, not the `while` loop, which continues along). The `switch` function just jumps to the appropriate label (recall that the colon indicates a label) and then picks up from there—and continues. Thus, if there were no `break` after `verbose++`, then the program would merrily continue on to execute `weighting_function`, and so on. This is called *fall-through*. There are reasons for when fall-through is actually desirable, but it seems to just be an artifact that `switch-case` is basically a smoothed-over syntax for using labels, `goto`, and `break`. The reader who uses `switch` statements will want to take care to have `break`s at the end of every `case`. The `break` at the end of the list of `case`s is extraneous, but there is a good chance that you will add to your list of `case`s, at which point the `break` will no longer be extraneous and will prevent a fall-through bug.

An alternative to the `switch` is a simple series of `if`s:

```
char c;
while ((c = get_opt(...))){
    if (c == 'v'){
            verbose++;
    } else if (c == 'w'){
            weighting_function();
    } else if (c == 'f'){
            fun_function();
    }
}
```

There is more typing and more redundancy here—the name `c` is repeated three times where in the `switch` setup it was used once—but there is no risk of fall-through mistakes.

As a concluding exercise, remember that I cut this segment from *Modeling with Data* because you can do all of these things using `if` statements. I showed you an alternative for `switch-case`; How you would implement the `goto`, `break` and `continue` examples with just `if` (and an extra variable or two)? When do you think the versions here or the `if` versions would be more readable?