SOME THOUGHTS ON LOOP CONSTRUCTS

Jules Jacobs

September 18, 2022

Programming languages have by now mostly settled on `for`, `foreach`, `while-do`, and `do-while` as loop constructs, often enriched with `break` and `continue` statements. While nobody questions this state of affairs any more, loop constructs used to be a subject of fierce debate. Knuth's Structured Programming with go to Statements [Knu74] is a great read if you want to know what alternative loop constructs looked like and the arguments for the various alternatives.

In this note I want to revisit the arguments for them with the benefit of modern hindsight, and look at some old proposals that did not make it but perhaps should have. Only minimal effort was made to make this coherent, so this is mostly a collection of separate little notes.

1 A MINIMAL BUT GENERAL LOOP CONSTRUCT

Assuming we want structured control flow and not `goto`, a minimal set of constructs that gets the job done is `loop { ... }`, which does an infinite loop, `break`, which jumps past the end of the loop, and `continue`, which jumps back to the start of the loop. We also need `if`, which we assume that we already have.

Wouldn't only `while` be more minimal? Yes, it would, but `while` alone is not sufficient to express all structured control flow without introducing auxiliary variables or code duplication. The combination of `loop` + `break` + `continue` *is* sufficient for that, provided our `break` and `continue` can jump up several levels of loops.

Rather than `loop`, we could also have `block { ... }`, which would by default jump out of the trailing curly brace if control flow reaches that point. We can express `loop` in terms of `block` and vice versa by adding a `break` or `continue` at the end. So we could choose to have either `loop`, or `block`, or both.

2 WHY `while`

That is the first argument for `while`: we get two constructs for the price of one by writing `loop` as `while(true)` and `block` as `while(false)`. This arguably improves clarity while losing almost nothing in terms of concision.

Of course this is not a terribly strong argument in favour of `while`. The stronger argument is that we would have to write `while(p) { ... }` as `loop { if(!p){ break; } ... }` if we only had `loop`. This is longer, has an awkward double negation, a nested `if`, and forces the programmer to mentally recognize this very common pattern and think "oh, that's a `while`-loop".

2.1 *Why not `do-while`*

What about `do-while`? In my opinion, the argument for including that is weak. To resolve this question with a poor man's experiment, I solved 100 leetcode puzzles. One has to use `while` all the time, but `do-while` came up only 2 times. In both cases no clarity was lost by writing it as either `while(true){ ... if(!p) break; }` or as `while(false){ ... if(p) continue; }`. In fact,

I've found it to be more common to have `break` or `continue` somewhere in the middle of the loop than at the end, so it seems silly to have special case syntax for the latter.

2.2 *Ole-Johan Dahl's proposal*

Knuth writes that Ole-Johan Dahl proposed the following: we have a `loop ... repeat` construct which is the same as our `loop { ... }`, and then simply make `while(p)` syntactic sugar for `if(!p){ break; }`. This allows us to write an ordinary while loop as `loop while(p) ... repeat` and a `do-while` loop as `loop ... while(p) repeat`. We can write `loop ... while(p) ... repeat` when the exit test is in the middle. There is much to like about this proposal: it teases the while loop apart into orthogonal pieces, and it gets rid of the double negation and nested `if` even for exit tests that appear in the middle.

3 WHY `for`

The loop `for(int i=0; i<10; i+=1){ ... }` can be written as `int i=0; while(i<10){ ...; i+=1; }`. Why should we prefer the `for` loop?

The problem with the `while` loop is that the different operations on `i` are spread out over different locations: the initialization is outside of the loop, the test is in the loop header, and the increment is all the way at the end.

The `for` loop has the advantage that it puts all of these together in the loop header, so that we can see at a glance that the loop body will be executed for `i = 0,1,...,9`.

Zig's while(i<10) : (i+=1)

3.1 *Inline initialization*

3.2 *Parallel assignment*

best = 0 while(...) best = max(best,x)

while(...) best = 0 then max(best,x)

while(...) best = phi(0,max(best,x))

for i=0 then i+1 while i<10

4 WHY `foreach`

5 MULTI-EXIT LOOPS

6 GOTOS WITH ARGUMENTS

7 RECURSIVE LOOPS

REFERENCES

[Knu74] Donald E. Knuth. Structured programming with go to statements. *ACM Comput. Surv.*, 6(4):261–301, dec 1974. doi:10.1145/356635.356640.