The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.8.6 Jump statements

jump statement syntax

jump-statement:

goto *identifier* **;**
continue ;
break ;
return $expression_{opt}$ **;**

Commentary

These are all jump statements in the sense they cause the flow of control to jump to another statement (in the case of the **goto** statement this could be itself).

Other Languages

Most imperative languages contain some form of **goto** statement, even those intended for applications involving safety-critical situations. Snobol 4 does not have an explicit jump statement. All statements may be followed by the name of a label, which is jumped to (it can be conditional on the success or failure of the statement) on completion of execution of the statement. The **come from** statement is described by Clark.[5]

A number of languages support some mechanism for early loop termination (e.g., Ada and Modula-2 support an **exit** statement). Some languages require the exit point to be labeled, others simply exit the loop containing the statement. Perl uses the keywords **next** and **last**, rather than **continue** and **break** respectively.

Common Implementations

processor pipeline

On most modern processors instruction execution is broken down into stages that are executed in sequence (known as an *instruction pipeline*). Optimal performance requires that this pipeline be kept filled with instructions. Jump statements (or rather the machine code generated to implement them) disrupt the smooth flow of instructions into the pipeline. This disruption occurs because the instruction fetch unit assumes the next instruction executed will the one following the current instruction, the processor is not aware it has encountered a branch instruction until that instruction has been decoded, by which time it is one or more stages down the pipeline and the following instruction is already in the pipeline behind it. Until the processor executes the branch instruction it does not know which location to fetch the next instruction from, a pipeline *stall* has occurred. Branch instructions are relatively common, which means that pipeline stalls can have a significant performance impact. The main techniques used by processor vendors to reduce the impact of stalls are discussed in the following C sentences.

One of the design principles of RISC was to expose some of the underlying processor details to the translator, in the hope that translators would make use of this information to improve the performance of the generated machine code. Some of the execution delays causes by branch instructions have been exposed. For instance, many RISC processors have what is known as a *delay slot* immediately after a branch instruction. The instruction in this delay slot is always executed before the jump occurs (some processors allow delay slot instructions following a conditional branch to be annulled). This delay slot simplifies the processor by moving some of the responsibility for keeping the pipeline full to the translator writer (who at worst fills it with a no-op instruction). Most processors have a single delay slot, but the Texas Instruments TMS320C6000[12] has five.

Fetching the instructions that will soon be executed requires knowing the address of those instructions. In the case of function calls the destination address is usually encoded as part of the instruction; however, the function return address is usually held on the stack (along with other housekeeping information). Maintaining a second stack, containing only function return addresses, has been proposed, along with speculative execution (and stack repair if the speculation does not occur along the control flow path finally chosen[11]).

Calder, Grunwald, and Srivastava[3] studied the behavior of branches in library functions, looking for common patterns that occurred across all calls.

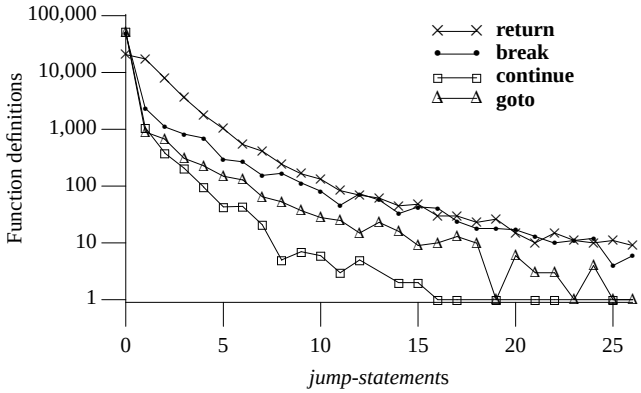

Figure 1782.1: Number of function definitions containing a given number of *jump-statement*s. Based on the translated form of this book's benchmark programs.

Coding Guidelines

The **continue** and **break** statements are a form of **goto** statement. Some developers consider them to be a *structured goto* and treat them differently than a **goto** statement. The controversy over the use of the **goto** statement has not abated since Dijkstra's, now legendary, letter to the editor was published in 1968.[6] Many reasons have been given for why source code should not contain **goto** statements; Dijkstra's was based on human cognition. Knuth argued that in some cases use of **goto** provided the best solution.[9]

Edsger W. Dijkstra

> *My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.*

The heated debate on the use of the **goto** statement has generated remarkably little empirical research.[8] Are guideline recommendations against the use of **goto** simply a hang over from the days when developers had few structured programming constructs (e.g., compound statements) available in the language they used, or is there a worthwhile cost/benefit in recommending against their use?

It is possible to transform any C program containing jump statements to one that does not contain any jump statements. This may involve the introduction of additional **while** statements, **if** statements, and the definition of new objects having a boolean type. An algorithm for performing this transformation, while maintaining the topology of the original flow graph and the same order of efficiency, is given by Ashcoft and Manna.[2] Ammaraguellat[1] gives an algorithm that avoids code replication and normalizes all control-flow cycles into single-entry single-exit **while** loops. In practice automated tools tend to take a simpler approach to transformation.[7] The key consideration does not appear to be the jump statement itself, but the destination statement relative to the statement performing the jump. This issue is discussed elsewhere.

1783 jump statement causes jump to

Usage

Numbers such as those given in Table 1782.1 and Table 1782.2 depend on the optimizations performed by an implementation. For instance, unrolling a frequently executed loop will reduce the percentage of branch instructions.

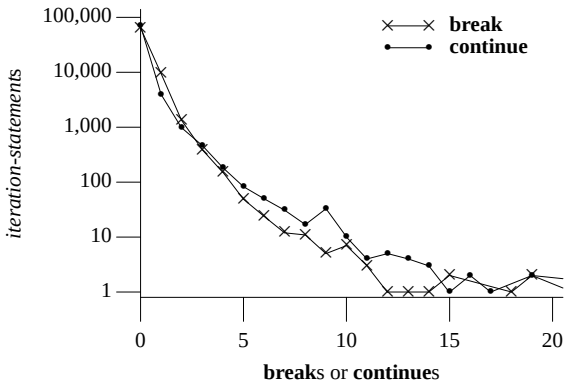

Figure 1782.2: Number of *iteration-statement* containing the given number of **break** and **continue** Based on the visible form of the `.c` files.

Table 1782.1: Dynamic occurrence of different kinds of instructions that can change the flow of control. *%Instructions Altering Control Flow* is expressed as a percentage of all executed instructions. All but the last row are expressed as percentages of these, control flow altering, instructions only. The kinds of instructions that change control flow are: conditional branches *CB*, unconditional branches *UB*, indirect procedure calls *IC*, procedure calls *PC*, procedure returns *Ret*, and other breaks *Oth* (e.g., signals and **switch** statements). *Instructions between branches* is the mean number of instructions between conditional branches. Based on Calder, Grunwald, and Zorn.[4]

Program	%Instructions Altering Control Flow	%CB	%UB	%IC	%PC	%Ret	%Oth	%Conditional Branch Taken	Instructions Between Branches
burg	17.1	74.1	6.9	0.0	9.5	9.5	0.0	68.8	7.9
ditroff	17.5	76.3	4.2	0.1	9.7	9.8	0.0	58.1	7.5
tex	10.0	75.9	10.7	0.0	5.8	5.8	1.9	57.5	13.2
xfig	17.5	73.6	7.7	0.6	8.6	9.2	0.3	54.8	7.8
xtex	14.1	78.2	8.5	0.2	6.0	6.2	1.0	53.3	9.1
compress	13.9	88.5	7.6	0.0	2.0	2.0	0.0	68.3	8.1
eqntott	11.5	93.5	2.1	1.5	0.7	2.2	0.0	90.3	9.3
espresso	17.1	93.2	1.9	0.1	2.3	2.4	0.1	61.9	6.3
gcc	16.0	78.9	7.4	0.4	6.1	6.5	0.8	59.4	7.9
li	17.7	63.9	8.7	0.4	12.9	13.2	0.9	49.3	8.9
sc	22.3	83.5	3.9	0.0	6.3	6.3	0.0	64.3	5.4
Mean	15.9	80.0	6.3	0.3	6.3	6.6	0.5	62.4	8.3

Table 1782.2: Number of static conditional branches sites that are responsible for the given quantile percentage of dynamically executed conditional branches. For instance, 19 conditional branch sites are responsible for over 50% of the dynamically executed branches executed by `burg`. *Static count* is the total number of conditional branch instructions in the program image. Of the 17,565 static branch sites, 69 branches account for the execution of 50% of all dynamic conditional branches. Not all branches will be executed during each program execution because many branches are only encountered during error conditions, or may reside in unreachable or unused code. Based on Calder, Grunwald, and Zorn.[4]

Program	10%	20%	30%	40%	50%	60%	70%	80%	90%	95%	99%	100%	Static count
burg	1	3	5	9	19	33	58	95	135	162	268	859	1,766
ditroff	3	11	19	28	38	50	64	91	132	201	359	867	1,974
tex	3	7	15	26	39	58	89	139	259	416	788	2,369	6,050
xfig	8	31	74	138	230	356	538	814	1,441	2,060	3,352	7,476	25,224
xtex	2	8	15	22	36	63	104	225	644	1,187	2,647	6,325	21,597
compress	1	2	2	3	4	5	6	8	12	14	16	230	1,124
eqntott	1	1	1	2	2	2	2	3	14	42	72	466	1,536
espresso	4	10	19	30	44	63	88	121	163	221	470	1,737	4,568
gcc	13	38	77	143	245	405	641	991	1,612	2,309	3,724	7,639	16,294
li	2	4	7	11	16	22	29	38	52	80	128	557	2,428
sc	2	3	4	6	9	16	30	47	76	135	353	1,465	4,478
Mean	3	10	21	38	62	97	149	233	412	620	1,107	2,726	7,912

Semantics

1783 A jump statement causes an unconditional jump to another place.

jump statement causes jump to

Commentary

For the **`goto`**, **`continue`**, and **`break`** statements the other place is within the function body that contains the statement.

Other Languages

Some languages allow labels to be treated as types. In such languages jump statements can jump between functions (there is usually a requirement that the function jumped to must have an active invocation in the current call chain at the time the jump statement is executed). In Algol 68 a label is in effect a function that jumps to that label. It is possible to call that function or take its address. In C terms:

```
somewhere:
    /* ... */
    if (problem)
       somewhere();    /* Same as "go to somewhere". */
    /* ... */
    void (*fp)(void) = &somewhere;
```

Common Implementations

The **`goto`**, **`continue`** and **`break`** statements usually map to a single machine instruction, an unconditional jump. Jumping out of a compound statement nested within another compound statement often creates a series of jumps. For instance, in the following example:

```
if (x)
   {
   for (;;)
      {
      some_code;
      if (y)
         break; /* Jump out of loop. */
      }
/* Place A */
   }
```

```
11  else
12     some_more_code;
13  /* Place B */
```

the branch instruction out of the loop, generated by the **break** statement, is likely to branch to an instruction (at place A) that branches over the **else** arm of the **if** statement (to place B). One of the optimization performed by many translators is to follow a *jump chain* to find the final destination. The destination of the branch instruction at the start of the chain being modified to refer to this place.

Many processors have span-dependent branch instructions. That is, a short-form (measured in number of bytes) that can only branch relatively small distances, while a long-form can branch over longer distances. When storage usage needs to be minimized it may be possible to use a jump chain to branch to a place using a short-form instruction, rather than a direct jump to that place using a long-form instruction (at the cost of reduced execution performance).[10]

Coding Guidelines

The term *spaghetti code* is commonly used to describe code containing a number of jump statements whose various destinations cause the control flow to cross the other control flows (a graphical representation of the control flow, using lines to represent flow, resembled cooked spaghetti, i.e., it is intertwined).

Jumps can be split into those cases where the destination is the most important consideration and those where the jump/destination pair need to be considered— as follows:

- *Jumping to the start/end of a function/block*— the destination being in the same or outer block relative to the jump . This has a straight-forward interpretation as restarting/finishing the execution of a function/block. The statement jumped to may not be the last (some termination code may need to be executed, or a guideline recommendation that functions have a single point of exit may cause the label to be on a **return** statement).
- *Jumping into a nested block.* This kind of jump/destination pair is the one most often recommended against.

goto EXAMPLE

iteration statement executed repeatedly

- *Jumping out of a nested block.* This kind of jump/destination pair may be driven by the high cost of using an alternative construct. For instance, adding additional flags to cause a loop to terminate may not introduce excessive complexity when a single nesting level is involved.
- *Jumping within the same block.* This is the most common kind of **goto** statement found in C source (see right plot of Figure 1783.2).

Jump statements written by the developer can create a flow of control that is that requires a lot of effort to comprehend. Some guideline documents recommend against the use of any jump statement (including a **return**, unless it is the last statement within a function). Comprehending the flow of control is an important part of comprehending a function. The use of jump statements can increase the cost of comprehension (by increasing the complexity of the flow of control) and can increase the probability that the comprehension process reaches the wrong conclusion (unlike other constructs that change the flow of control, there are not usually any addition visual clues, e.g., indentation, in the source signaling the presence of a jump statement). However, there is no evidence to suggest that the cost of the alternatives is any lower and consequently no guideline recommendation is made here.

dead code

Any statements that appear after a *jump-statement*, in the same block, are dead code.

Usage

A study by on Gellerich, Kosiol, and Ploedereder[8] analyzed **goto** usage in Ada and C. In the translated form of this book's benchmark programs 20.6% of **goto** statements jumped to a label that occurred textually before them in the source code.

footnote 134

134) Thus, *clause-1* specifies initialization for the loop, possibly declaring one or more variables for use in the loop;

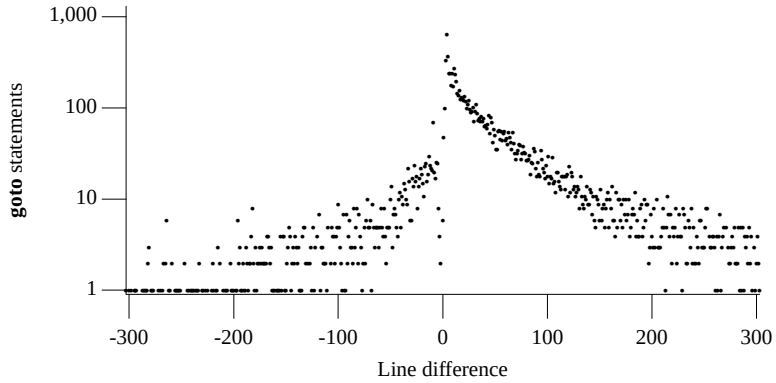

Figure 1783.1: Number of **goto** statements having a given number of visible source lines between a **goto** statement and its destination label (negative values denote backward jumps). Based on the translated form of this book's benchmark programs.

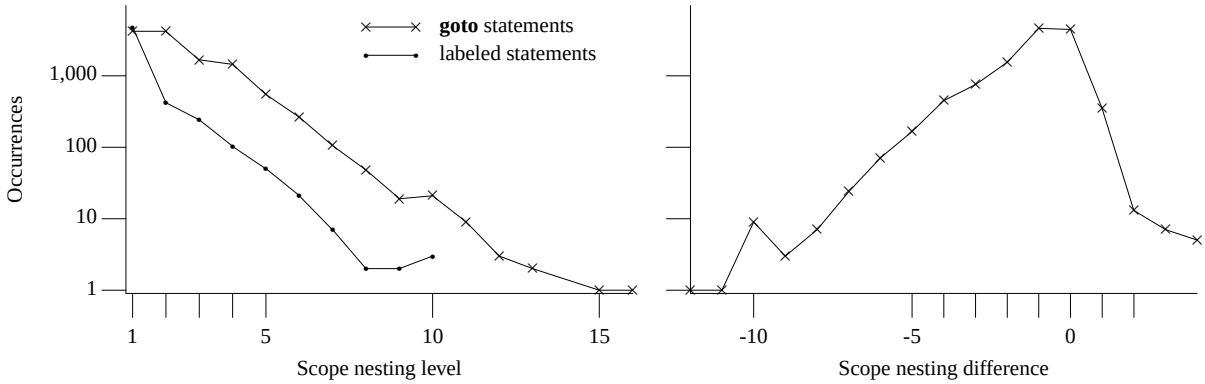

Figure 1783.2: Number of **goto** statements and labels having a given scope nesting level (nesting level 1 is the outermost block of a function definition), and on the right the difference in scope levels between a **goto** statement and its corresponding labeled statement (negative values denote a jump to a less nested scope). Based on the translated form of this book's benchmark programs.

Commentary

Technically the evaluation *clause-1* need have no connection with the contents of the body of the loop. This wording expresses the view of C committee about how they saw this construct being used by developers.

C90

Support for declaring variables in this context is new in C99.

C++

The C++ Standard does not make this observation.

1785 the controlling expression, *expression-2*, specifies an evaluation made before each iteration, such that execution of the loop continues until the expression compares equal to 0;

Commentary

In C90 this wording appeared in a footnote, where it was an observation about the equivalence between a **for** statement and a **while** statement. In C99 this wording plays the role of a specification.

Coding Guidelines

The discussion on the controlling expression in an **if** statement is applicable here.

controlling expression for statement

1786 and *expression-3* specifies an operation (such as incrementing) that is performed after each iteration.

Commentary

The *incrementing* operation referred to here is the concept of a loop control variable having its value increased (often by one). Decrementing (i.e., decreasing the value of the loop control variable) is a less commonly specified operation (as is walking a linked list). There is often a causal connection between the operand that is incremented and one of the operand appearing in *expression-2*.

References

1. Z. Ammaraguellat. A control-flow normalization algorithm and its complexity. *Software Engineering*, 18(3):237–251, Mar. 1992.
2. E. A. Ashcroft and Z. Manna. The translation of 'go to' programs to 'while' programs. Technical Report CS-TR-71-188, Stanford University, Department of Computer Science, Jan. 1971.
3. B. Calder, D. Grunwald, and A. Srivastava. The predictability of branches in libraries. Technical Report Research Report 95/6, Western Research Laboratory - Compaq, 1995.
4. B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2(4):313–351, 1995.
5. R. L. Clark. A linguistic contribution of GOTO-less programming. *Datamation*, 19(12):62–63, Dec. 1973.
6. E. W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.
7. A. M. Erosa. A goto-elimination method and its implementation for the McCat C compiler. Thesis (m.s.), McGill University, Montreal, Canada, May 1995.
8. W. Gellerich, M. Kosiol, and E. Ploedereder. Where does GOTO go to? In *Reliable Software Technology —Ada-Europe 1996*, volume 1088 of *LNCS*, pages 385–395. Springer, 1996.
9. D. E. Knuth. Structure programming with `go to` statements. *Computing Surveys*, 6(4):261–301, Dec. 1974.
10. B. Leverett and T. G. Szymanski. Chaining span-dependent jump instructions. *ACM Transactions on Programming Languages and Systems*, 2(3):274–289, 1980.
11. K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *Proceedings of the* 31^{st} *Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-98)*, pages 259–271, Los Alamitos, Nov. 30–Dec. 2 1998. IEEE Computer Society.
12. Texas Instruments. *TMS320C6000 CPU and Instruction Set Reference Guide*. Texas Instruments, spru189f edition, Oct. 2000.