Go To Statement Considered Harmful: A Retrospective

David R. Tribble
Revision 1.1, 2005-11-27

Introduction

This is a discussion and analysis of the letter sent to *Communications of the Association for Computing Machinery* (CACM) in 1968 by **Edsger W. Dijkstra**, in which he calls for abolishing the **goto** statement in programming languages.

The letter has become quite famous (or infamous, depending on your feelings about goto statements) in the 40 years since it was first published, and is probably the most often cited document about any topic of programming. It is also probably the least read document in all of programming lore.

Most programmers have heard the adage *"Never use goto statements"*, but few of today's computer science students have the benefit of the historical context in which Dijkstra made his declaration against them. Modern programming dogma has embraced the myth that the goto statement is evil, but it is enlightening to read the original tract and realize that this dogmatic belief entirely misses the point.

This paper was written at a time when the accepted way of programming was to code iterative loops, **if-then**s, and other control structures by hand using goto statements. Most programming languages of the time did not support the basic control flow statements that we take for granted today, or only provided very limited forms of them. Dijkstra did not mean that *all* uses of goto were bad, but rather that superior control structures should exist that, when used properly, would eliminate *most* of the uses of goto popular at the time. Dijkstra still allowed for the use of goto for more complicated programming control structures.

Background

It cannot be overstated the importance of the work that Dijkstra and others (C. A. R. Hoare, Niklaus Wirth, et al) provided in the early days to the fledgling discipline of computer programming. Their contributions were extremely instrumental in establishing computer science as a rigorous discipline in and of itself and algorithmic programming as an official branch of mathematics and logic.

Dijkstra, like most of his colleages in the early formative days of serious computer science, was an academic with heavy mathematical training. Not surprisingly, then, much of the early work in computer science was undertaken with the goal of making computer programming a rigorous engineering discipline with a solid foundation in mathematics and logic. The hope was that programming languages could be developed that made it possible to prove the correctness of programs. The theory, called *formal verification*, was that a small set of programming constructs (**if-then-else** and looping statements, primitive data types, as so forth) could be devised that would be sufficiently powerful to make it possible to define any possible programming task, and which could be mathematically proven to be correct (i.e., have no logic errors).

This movement, which began in the late 1950s, was similar in spirit to the earlier movement in mathematics known as *Hilbert's programme*, expressed by David Hilbert, which was intended to codify all of mathematics in a complete and all-inclusive set of laws derived from nothing more than the building blocks of the natural numbers and the rules of simple logic and arithmetic. Alas, Kurt Gödel's Incompleteness Theorem deflated that dream, proving that there are mathematical truths (and untruths) that are outside the realm of logical provability.

Another parallel can be drawn between the formal verification movement and Newtonian physics. In the early days following the acceptance of Newton's Laws of Motion, many aspired to the belief that the physical universe was deterministic, and that all motion and activity could eventually be calculated to arbitrary precision given enough knowledge beforehand of the masses and momentums involved. Alas, the advent of quantum mechanics brought about by the discoveries of Heisenberg, Bohr, and others put an end to that belief with the realization that deep down at the particle level all physical behavior is inherently probabilistic and random, and therefore unpredictable.

In a similar vein, the goals of formal verification were eventually seen as unworkable. Dijkstra later abandoned the search for program provability and turned instead to the study of techniques for correct *program derivation*. Such techniques were designed to allow a programmer to construct programs in a methodical fashion that guaranteed that they exhibited correct behavior. This area of study shares much in common with the techniques of *top-down design* and *functional decomposition*.

It should also be realized that much of the terminology of programming that is taken for granted today had not yet been firmly established in 1968. There was much debate and discussion at the time about what terms to use for programming concepts, and most of the terms we use today took many years to be widely accepted.

Dijkstra was very much an academician, tending to use technically laden verbiage in his writings. This may explain, to some degree, why his famous "Go To" letter has been read by so few people.

It is noteworthy to point out that he despised the term *bug* to denote a programming mistake, preferring instead to use the term *error*. Today, of course, the two terms are still both used in the context of programming, but whereas *error* means (as it always has) a mistake produced by either a person or a machine, the term *bug* has come to mean something more specific, applied only to the realm of man-made systems, particularly programmable computers, to denote a specific failure in design or unexpected execution result. The term *debug* has also sprung into existence to denote the specific activity of finding and removing bugs from a system, a nice term that would not have been invented if we had been left with only the word *error*.

Part I

Dijkstra's Letter, Annotated

What follows is Dijkstra's famous "Go To" letter to CACM in 1968, along with annotations that discuss the details of the letter from a historical perspective.

Go To Statement Considered Harmful

Edsger W. Dijkstra

Reprinted from *Communications of the ACM*,
Vol. 11, No. 3, March 1968, pp. 147-148.

Annotations by
David R. Tribble

Editor:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

Dijkstra introduces the topic of his letter, which is that he has noticed that goto statements are mostly detrimental to the programs in which they appear. He posits that the more gotos a programmer uses, the worse a programmer he is.

He proposes that goto statements should be abolished from all high-level programming languages. He even hints that goto should be eliminated from *all* programming languages, perhaps including machine code, although one is left to wonder exactly how this could be accomplished.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

This paragraph of technically dense verbiage is fairly typical of Dijkstra's academic writing style.

This just means that the actual activity performed by a programmer is not simply writing programs, but controlling the action of the code as it is executed on an actual machine. However, he states, once the programmer has written a working program, the actual execution of the program is entirely under the control of the machine itself.

Dijkstra uses the term *correct* to describe a program that has no errors, or in current parlance, has no bugs. This terminology reflects the belief at the time that code could be written which could be *formally verified*, i.e., that such code could be subjected to a series of mathematical and logical manipulations that would demonstrate that the code either contained errors (logic errors, constraint errors, invariant errors, etc.) or that it did not and thus was thus *provably correct*. As mentioned in the Background section above, computer scientists (or at least programmers) do not think about programming in such terms today.

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.

Here Dijkstra observes that humans are better at visualizing static relationships than dynamic relationships. Thus, he argues, we should minimize the difference between the two when expressed as program code, so that the dynamic (nonconstant) aspects of the program are evident in the structure of the source code itself.

This is generally true in most current programming languages, the majority of which operate in a linear, statement-by-statement fashion. However, to some extent Dijkstra's principle has not been fully realized when we observe the complexity that must be dealt with by real-world programming tasks, such as multitasking, multithreading, interrupt handling, volatile hardware registers, virtual memory paging, device latency, real-time event handling, and so forth, just to name a few. Today's programming problems are no longer sufficiently handled by simple one-line-at-a-time execution programming models.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion regarded as the descriptions of single actions) it is sufficient to point in the program text to a point between two successive action descriptions.

Dijkstra begins to construct a formal definition of *program execution*, or what he calls *progress of a process*. The discussion that follows is similar to the definition of *sequence points* used in the formal definition of the execution model employed by the C and C++ (and other) languages.

It must be remembered that many of the terms we take for granted today were not firmly established place at the time, and there was no commonly accepted language or pseudo-language in use for discusing algorithms and programs. Today, of course, a writer would use a concrete language such as C, Java, Pascal, LISP, or a pseudo-language bearing a strong resemblance to one of these languages as a *lingua franca* for illustrating programming concepts.

(In the absence of **go to** statements I can permit myself the syntactic ambiguity in the last three words of the previous sentence: if we parse them as "successive (action descriptions)" we mean successive in text space; if we parse as "(successive action) descriptions" we mean successive in time.) Let us call such a pointer to a suitable place in the text a "textual index."

In typical academic style, Dijkstra employs a bit of linguistic cleverness to allow the ambiguous phrase *successive action descriptions* to take on two different meanings. This reflects the dual nature of programming tasks that he mentioned previously, these being related to the sequential nature of executing one statement (or action) after another, i.e., allowing source code for a program to be composed of a series of separate statements (actions) to reflect the sequential nature in which the statements are to be executed in time.

His term *textual index* is essentially a *program counter*. However, he is attempting to go beyond simply tracking the location of the current execution thread, to making an explicit connection between a statement in the source code text and a program execution state. So perhaps a better term would be *statement pointer*.

When we include conditional clauses (**if** B **then** A), alternative clauses (**if** B **then** A_1 **else** A_2), choice clauses as introduced by C. A. R. Hoare (**case**[i] **of** $(A_1, A_2, \cdots, A_n)$), or conditional expressions as introduced by J. McCarthy $(B_1 \rightarrow E_1, B_2 \rightarrow E_2, \cdots, B_n \rightarrow E_n)$, the fact remains that the progress of the process remains characterized by a single textual index.

Dijkstra introduces more complex flow control statements such as **if-then-else** conditional statements and **case** (a.k.a. **select** or **switch**) selection statements, noting that these do not change the basic nature of his *textual index* (or *statement pointer*).

This reflects the fact that, at the time, much effort was being made to formulate the best minimal set of flow control structures for programming languages and for programming theory in general. It is no accident that most of the constructs resembled the control structures supported by ALGOL, because many of the people who worked or influenced the design of ALGOL were academics who wrote a great deal about programming.

A major goal of all of this effort was to create a nomeclature that could be used not just for actual programming languages, but which also could be used directly for mathematical formulations of programming algorithms. As stated in the Background section above, this reflected a belief that programs could be expressed in a form that would make it possible to prove their correctness mathematically.

As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points to the interior of a procedure body the dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

This is an observation that a single statement pointer is not sufficient to define the state of an executing program if the program employs *subroutines* (variously known as *procedures*, *functions*, or *methods*). To handle this additional complexity, Dijkstra defines a *sequence of textual indices*.

This reflects what is known in modern parlance as a *call stack*, which is an array of program counters (a.k.a. *return addresses*), each designating the last statement from which a procedure call was made. Since he is establishing an explicit relation between a textual index and the program execution state, though, it would be more correct to think of the call stack as an array of statement pointers. The number of statement pointers needed is simply the number of procedure calls that are currently active at a given point in the execution, i.e., the depth of the call stack.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses.

Dijkstra adds repetition control flow statements to the mix. He states in passing that such repetition statements are completely unnecessary because they can be replaced with equivalent recursive calls.

This reflects the fact that, at that time, recursion was very much in vogue and was considered by many, especially the more academically inclined, to be a superior form of expressing programs and algorithms. The reason for this popularity is that recursive definitions have a history of mathematical rigor - specifically, *recursive formulas* and *recurrence relations*, which deal with recursively defined sequences wherein each element in the sequence is defined in simpler terms using previous elements in the sequence. Two classic examples are the factorial function, $n! = n(n-1)!$, and the Fibonacci sequence, $F_i = F_{i-2} + F_{i-1}$.

This is a typically academic observation. It is true in theory that any looping statement can be replaced with a recursive call, and certain languages such as LISP do in fact support a recursive style of programming (also called *functional programming*). However, for most programming applications, and consequently what is actually supported by most programming languages, recursion plays only a minor (but still very useful) role.

Dijkstra mentions that iterative statements can be implemented on *finite equipment*, which of course is what all actually existing machines are, no matter how much virtual memory they possess. This is a subtle way of admitting that some forms of recursion require potentially infinite resources (i.e., an infinite call stack). Consider a typical embedded application having a main program loop that polls for an event, processes the event, then waits for the next event, looping forever. Such an infinite loop could indeed be written as a tail-recursive procedure call, but what would be the point? More complicated forms, which are written recursively simply for the sake of being recursive, would be impossible to program on most real systems.

Dijkstra seems to imply that iterative looping (inductive) statements are intellectually harder to grasp than recursion, which is the kind of thing only a mathematician would say.

iterate, v. - See *iterate*.

With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

With the addition of repetition control structures, we require a way of specifying more than just the current statement - we also need to keep track of which *iteration* of each loop is currently being executed. So just like nested procedure calls, we must employ a *loop iteration stack* to track these iteration counts, with one entry per (nested) loop, which Dijkstra calls a *dynamic index sequence*.

So, putting it all together, we have a *textual index sequence* (*call stack*) and a *dynamic index sequence* (*loop iteration stack*), which together define the current state of an executing program.

(Nestedly?)

The main point is that the values of these indices are outside the programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Again, Dijkstra states the obvious, that once a program is written and running, the programmer no longer has any control over the actual execution. The execution is represented by the contents of the call stack and loop iteration stack at any given point during the execution - what Dijkstra calls the *independent coordinates* of the program execution, and what we might simply call the *state* or *execution history* of the program.

Why do we need such independent coordinates? The reason is - and this seems to be inherent to sequential processes - that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, *n* say, of people in an initially empty room, we can achieve this by increasing *n* by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of *n*, its value equals the number of people in the room minus one.

Dijkstra observes that the value of a given variable in a program can only be known if the history of the program's execution is known precisely up to a given point in time. In other words, a program is expected to execute in a *deterministic* fashion, and it should be possible to determine the value of any variable at any point during the execution from the history of the execution (or the history of the program states) up to that point.

Dijkstra introduces the concept of an *in-between moment* prior to the completion of a program statement. This is similar to the notion of a *sequence point* as specified in languages like C and C++, which defines precisely when actions are to occur and in what order, and just as importantly, what actions are left unspecified.

The execution of a program is well-defined only at specific sequence points, which typically occur at the end of statements, prior to function calls, and at specific points in the evaluation of subexpressions. Between any two sequence points, the state of the program is not well-defined, which means that until the next sequence point is reached, the values of the program variables are in an indeterminate (or *in-between*) state.

Dijkstra suggests a simple example of this: incrementing a counter, or a statement such as **n = n+1**. When this statement is actually executed, there is a point during which the previous value of **n** is read and **1** has been added to it, but that new value has not yet been written back to the variable **n**. This is the *in-between* state he refers to, or a state in the execution between two *sequence points*, during which the variable **n** still contains its old value instead of its new value.

The unbridled use of the **go to** statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the **go to** statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, *n* equals the number of persons in the room minus one!

Here, finally, we get to the crux of Dijkstra's argument concerning the lowly goto statement. Essentially, Dijkstra argues that the "unbridled use" of goto statements in a program obscures the execution state and history of the program, so that at any given moment the values of the call stack and loop iteration stack are no longer sufficient to determine the value of the program variables.

This obfuscation is a consequence of the fact that an unconstrained goto statement can transfer control out of a loop before it is completed, and likewise can transfer control into the middle of a loop that is already being iterated. Both cases complicate the way in which the counters in the loop iteration stack are modified.

Add to that the possibility of *non-local* gotos, which are transfers of control out of currently executing procedures back into previously called procedures, which really disrupt the execution state by invalidating the values of entire portions of the call stack.

Dijkstra states a specific example of transferring control out of a loop or procedure during an *in-between moment*, which renders the execution state indeterminate from that point on.

Another way to say this is that gotos can invalidate the *program invariants* that are supposed to be guaranteed inviolate by the program structure. The example invariant he uses here is that counter **n** always represents the number of persons in a room. Allowing unstructured gotos to change the course of execution could cause that invariant to become invalid (i.e., no longer invariant), thus making the value of **n** meaningless, or at least make it extremely difficulty to determine its true value from the execution history.

The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

What Dijkstra means by the goto statement *as it stands* is otherwise known as an *unstructured* goto. That is, a goto statement with no restrictions about how it may be used in an otherwise structured language.

Limiting the use of gotos to a few simple, well-structured controls such as exiting early from loops, error handling (a.k.a. exceptions), and the like brings the goto statement back into the realm of *structured* control flow modification.

But without rules that enforce these limitations, the goto statement provided by a language cannot be said to be truly *well-structured*.

Dijkstra admits that not all of the flow control structures provided by a language will satisfy all programming needs. This implies that goto still has its place for those fairly rare programming situations that require more complicated flow control.

Dijkstra mentions *abortion clauses*, or what is now commonly known as *exception handlers*, hinting that these kinds of things are really fancy gotos underneath, but that they can be defined so that they behave well within the confines of a structured language, i.e., that they will not corrupt the execution state in haphazard ways.

The exception handling clauses of object-oriented languages like Ada, C++, Java, and others have for the most part adhered to this principle, so that when an exception is thrown in these languages, the execution state (which includes global and local variables, the procedure call stack, the heap, etc.) is altered in clean and predictable ways.

More primitive languages such as FORTRAN, COBOL, C, Pascal, and the like may provide some kinds of primitive exception handling mechanisms, but their use does not guarantee that the execution state is cleanly preserved or that allocated resources will be properly released.

It is hard to end this with a fair acknowledgment. Am I to judge by whom my thinking has been influenced? It is fairly obvious that I am not uninfluenced by Peter Landin and Christopher Strachey. Finally I should like to record (as I remember it quite distinctly) how Heinz Zemanek at the pre-ALGOL meeting in early 1959 in Copenhagen quite explicitly expressed his doubts whether the **go to** statement should be treated on equal syntactic footing with the assignment statement. To a modest extent I blame myself for not having then drawn the consequences of his remark.

Here Dijkstra acknowledges the influences that led him to his observation about the dangers of goto. As noted in the Background section, many of the people who influenced the design of the ALGOL language were also involved in discussions about proper language design and correct program control flow structures.

The remark about the undesirability of the **go to** statement is far from new. I remember having read the explicit recommendation to restrict the use of the **go to** statement to alarm exits, but I have not been able to trace it; presumably, it has been made by C. A. R. Hoare. In [1, Sec. 3.2.1] Wirth and Hoare together make a remark in the same direction in motivating the **case** construction:

"Like the conditional, it mirrors the dynamic structure of a program more clearly than **go to** statements and switches, and it eliminates the need for introducing a large number of labels in the program."

Dijkstra states that goto statements may be deemed acceptable (only) for *alarm exits*, which we would call *fatal exceptions*. For languages that lack robust exception handling mechanisms, gotos may be the only practical substitute.

Dijkstra mentions the design of the **case** (or **select**) control flow structure as proposed by Hoare and Wirth. We take this control structure for granted today, but at the time its merits were still being debated. Dijkstra reminds us that it was originally proposed as a superior alternative to the clumsy use of multiple **if**s, **goto**s, and labels.

In [2] Guiseppe Jacopini seems to have proved the (logical) superfluousness of the **go to** statement. The exercise to translate an arbitrary flow diagram more or less mechanically into a jump-less one, however, is not to be recommended. Then the resulting flow diagram cannot be expected to be more transparent than the original one.

Dijkstra mentions *flow diagrams*, which reflects the state of the art of program design at the time. Since then, programming technology has evolved through the stages of structured programming, top-down programming, object-oriented programming, component programming, aspect programming, and beyond. And yet, in spite of all of these adavances in design, some of Dijkstra's main point about unstructured program flow remains just as valid now as then.

It must be noted that Dijkstra's final comment on the subject seems to imply that completely removing all of the gotos from one's own programs is a bad idea. While he states that it has been proven that goto statements are in fact redundant in any given program, Dijkstra nevertheless admits that removing *all* of the gotos in a program will render its flow more difficult to understand.

He is in fact arguing that *some* gotos in a program may be useful and may actually make the program easier to understand. So it is safe to say that Dijkstra considered goto statements to be *harmful*, but not *lethal*, and certainly not *useless*.

References:

1. Wirth, Niklaus, and Hoare C. A. R.
A contribution to the development of ALGOL.
Comm. ACM 9 (June 1966), 413-432.

2. Böhm, Corrado, and Jacopini Guiseppe.
Flow diagrams, Turing machines and languages with only two formation rules.
Comm. ACM 9 (May 1966), 366-371.

Edsger W. Dijkstra
Technological University
Eindhoven, The Netherlands

Part II

Structured Programming

Have programming languages evolved since Dijkstra's letter was published to the point that **goto** is no longer needed?

Since the 1960s, several advances in programming theory have occurred. The discipline of programming has progressed through several phases, with each new advance being touted as the next "better" way of programming. A brief list of some of these advancements:

- *structured programming*
- *functional decomposition*
- *top-down design and step-wise refinement*
- *bottom-up design*
- *iterative design*
- *third generation languages*
- *fourth generation languages*
- *fifth generation languages*
- *object-oriented programming*
- *components*
- *programming patterns*
- *aspect programming*

Each approach caused a paradigm shift in programming theory, affecting the way programmers wrote their programs on a day-to-day basis as well as changing the way programming languages were designed and the features they provided. But all of these advancements affected the structure of programs at levels above that of simple execution statements, i.e., at levels involving procedures, data objects, program modules, etc. The fundamental approach of programming at the lowest level, at the sequential *statement* level, is still the same as it was back in the early days of the first programming languages such as FORTRAN and COBOL.

The following sections describe the program flow constructs that are commonly available in most of today's programming languages. These have remained pretty much the same since their introduction at the advent of structured programming. (The constructs are shown in a pseudo-language instead of any specific language.)

---

if-then-else

All structured programming languages provide some form of the **if-then** flow control construct:

```
if conditional_expression then
    statement_1
```

and the **if-then-else** construct:

```
if conditional_expression then
    statement_1
else
    statement_2
```

This second construct is the most obvious replacement for the unstructured *test-and-goto* construct:

```
if conditional_expression then
    statement_1
    goto endif1
else1:
    statement_2
endif1:
    ...
```

The **if-then** statement can be implemented in machine code as something like the following:

```
# if-then statement
    move expression, reg1
    jump not condition, label1
    statement_1
label1:
    ...
```

And likewise for the **if-then-else** statement:

```
# if-then-else statement
    move expression, reg1
    jump not condition, label1
    statement_1
    jump label2
label1:
    statement_2
label2:
    ...
```

A common programming idiom is to write multiple **if-then** statements in a sequence:

```
if condition_1 then
    statement_1
else if condition_2 then
    statement_2
else if condition_3 then
    statement_3
else
    statement_4
```

It is more difficult in some languages (especially some older languages) to write multiple **if-then** sequences, so the construct ends up looking like the following code, which is functionally equivalent but harder to read:

```
if condition_1 then
    statement_1
else
    if condition_2 then
        statement_2
    else
        if condition_3 then
            statement_3
        else
            statement_4
        end
    end
end
```

Some languages provide a separate keyword for the **else-if** combination (variously called **elseif**, **elsif**, and **elif**), but the effect is the same. Some languages also provide a separate keyword for the last **else** clause in a multiple **if-then** sequence (e.g., **otherwise** or **default**).

select

Most structured programming languages provide some kind of multiple selection control construct (variously known as **case**, **select**, **switch**, **examine**, **inspect**, **choose**, **when**, etc.). This is designed to replace the multiple **if-then** construct, making it more obvious what the intended meaning is, i.e., selecting one of several choices for a given expression value:

```
select expression in
    case constant_1:
        statement_1

    case constant_2:
        statement_2
    case constant3:
        statement3

    default:
        statement4
end
```

This is equivalent to the sequence of multiple **if-then** statements shown above, with the **default** selection acting the part of the last **else** clause. Some older languages do not provide a **default** or **otherwise** clause.

The **select** statement can be implemented in machine code as something like the following:

```
    # select statement
        move expression, reg1
        cmp  reg1, constant1
        jump not equal, label1
        statement1
        jump label4
    label1:
        cmp  reg1, constant2
        jump not equal, label2
        statement2
        jump label4
    label2:
        cmp  reg1, constant3
        jump not equal, label3
        statement3
        jump label4
    label3:
        statement4
    label4:
        ...
```

More efficient implementations are possible, such as using an index into a jump table, or rearranging the comparisons to emulate an unrolled binary search, etc. Some CPUs provide special instructions for implementing **select** statements directly in machine code by utilizing a small jump table.

Some languages (notably C, C++, Java, C#, and other languages derived from C) allow for more complicated control flow by allowing each **case** statement to "fall through" to the next **case** clause. Purists say this sullies an otherwise logically clean control construct, while pragmatists say that it allows for more efficient code in many difficult programming situations.

do-while

This is the simplest form of iteration provided by most structured programming languages. Two variants are generally provided, one with the conditional test before the loop body (providing at least one iteration):

```
do
    statements
while conditional_expression
```

The other form places the conditional test after the loop body (providing zero or more iterations):

```
while conditional_expression do
    statements
end
```

These constructs are functionally equivalent to the following code that uses goto:

```
# do-while
loop:
    statements
    if conditional_expression
        goto loop
```

And:

```
# while-do
loop:
    if not conditional_expression
        goto endloop
    statements
    goto loop
endloop:
    ...
```

These constructs can be implemented in machine code as something like the following:

```
# do-while statement
label1:
    statements
    move expression, reg1
    jump condition, label1
    ...

# while-do statement
label1:
    move expression, reg1
    jump not condition, label2
    statements
    jump label1
label2:
    ...
```

Some languages provide other variants of the **do-while** statement, e.g., **do-until** (which reverses the sense of the conditional expression).

Some languages allow for more complicated flow control by providing clauses for terminating a loop iteration early (a **break** statement) or skipping the rest of the loop body and forcing the next loop iteration (a **continue** statement). These constructs allow for the *loop-and-a-half* situations that sometimes arise (which is discussed further below).

Some languages allow breaking out of nested loops with a **labeled break** construct. (This is discussed further in Example L-1 and Example N-1 below.)

for-loop

Most structured programming languages provide a more complex iteration construct that allows a counter or array index to be incremented or decremented with each iteration. This construct is functionally equivalent to a **do-while** loop, but provides a clearer intent of the controlling entity (the counter or index) of the loop iterations:

```
for i = low_value to high_value by increment do
    statements
end
```

This is equivalent to the following unstructured code that uses goto:

```
    i = low_value
loop:
    if i > high_value
        goto endloop
    statements
    i = i + increment
    goto loop
endloop:
    ...
```

This construct can be implemented in machine code as something like the following:

```
# for-loop statement
    move low_value, reg1
label1:
    cmp  reg1, high_value
    jump greater, label2
    statements
    add  increment, reg1
    jump label1
label2:
    ...
```

To fully support the **for-loop**, a language should handle negative increment values as well.

The form of the **for-loop** shown above iterates the body of the loop zero or more times. Other **for-loop** variants test the control variable (a.k.a. the *loop index*) after the loop body, which makes the loop iterate at least once.

Other variants of the **for-loop** allow more than one loop counter (or *loop index*) to be specified.

A common programming problem is handling *loop-and-a-half* constructs, i.e., a loop that must be terminated upon some condition in the middle of its loop body, so that some portion of the loop body is not executed during the last iteration. The following code uses a **break** statement to do this:

```
for i = low_value to high_value by increment do
    statements_1
    if condition
        break
    statements_2
end
```

Some languages provide special syntax specifically for terminating a loop in the middle of its body:

```
for i = low_value to high_value by increment do
    statements_1
exit when condition
    statements_2
end
```

Like the **break** statement, the **exit when** clause allows the loop to be terminated after the first half of its body has been executed but before the rest of the body is executed. Both forms replace the use of an explicit goto, as shown in the following code:

```
for i = low_value to high_value by increment do
    statements_1
    if condition
        goto endloop
    statements_2
end
endloop:
    ...
```

A related programming problem is coding a loop to execute a portion of its body and then skip the rest if some condition occurs, forcing the next iteration of the loop. Some languages provide a **continue** statement for this:

```
for i = low_value to high_value by increment do
    statements_1
    if condition
        continue
    statements_2
end
```

This replaces the use of an explicit goto, as shown in the following code:

```
for i = low_value to high_value by increment do
    statements_1
    if condition
        goto nextloop
    statements_2
```

```
        nextloop:
        end
```

Other Approaches

Several structured programming languages do not provide goto statements at all, including Modula-2, Modula-3, Oberon, Eiffel, and Java, on the assumption that the other flow control mechanisms they do provide are sufficient for all programming tasks and thus goto statements should never be needed. This is not always a good assumption to make when designing a programming language. Language designers cannot anticipate all possible programming scenarios, and providing an "escape mechanism" out of the normal control structures gives the programmer the ability to program around the syntactic limitations imposed by the language when the need arises. (Some examples of inadequate flow control constructs are discussed in more detail below.)

It is also worth noting that there are programming languages that do not provide structured flow control constructs at all. Many *functional programming* languages, such as LISP, Scheme, and Prolog, do not provide traditional structured flow control constructs beyond **if-then-else** or only provide very simple forms of them. Iteration in such languages is generally accomplished using some form of recursion, and these languages are in fact tailored specifically for recursive tasks and data structures. But what such languages lack in structured flow statements, they generally make up for by providing powerful *dynamic programming* mechanisms and extremely flexible non-homogeneous data types.

Part III

Is Goto Still Necessary?

Good programming language design dictates that a language should provide a sufficiently complete and powerful set of flow control constructs in order to make it relatively easy to write efficient code for any programming task. A language should not be overly ambitious by providing too many different ways to do the same thing, while at the same time it should not be anemic by providing too few ways for programming ideas to be expressed. The set of flow control statements and clauses provided should be powerful and flexible enough so that a programmer can express his ideas clearly and succintly without having to resort to the use of extraneous control variables or to rearrange his code unnaturally just to get around the syntactical restrictions of the language.

The following sections discuss two major programming problems that traditionally have been solved by using goto statements. These problems are presented in the light of current programming techniques, with an eye to seeing if the programming languages currently available are sufficiently advanced to handle them without using goto.

Loop Exits

Dijkstra's call for the complete elimination of goto statements is fine in theory, but would it work in practice? The control flow statements described above are sufficient for most programming logic, but there are programming situations that require more powerful constructs.

A common use for goto statements is to exit early from within loops, especially if the exit must break out from within two or more levels of nested loops. C provides simple loop escape mechanisms, i.e., the **break** and **continue** statements.

Example L-1 - Loop Exit Using Break

```
// Simple loop with an early exit

for (;;)
{
    int   ch;

    ch = read();
    if (ch == EOF)
        break;                  // With a loop escape

    parse(ch);
}
```

In order to exit a loop early without using such mechanisms, a trade-off must be made in which an additional flag (boolean) variable to signal completion of the loop is used. This incurs the overhead of an extra variable and an extra test of that variable at the top of each loop.

Example L-2 - Loop Exit Without Using Break

```
// Simple loop with no loop escape mechanism

bool  incomplete = true;

while (incomplete)
{
    int   ch;

    ch = read();
    if (ch == EOF)
        incomplete = false;     // Without a loop escape
    else
        parse(ch);
}
```

For more complicated exits from loops, namely breaking out of a loop iteration that is nested within two or more levels of loops, more extensive support is required of the language. Some languages, such as Java, provide the capability of breaking out of loops that are prefixed with label names.

Example N-1 - Exiting a Nested Loop

```
// [Java]
// Exiting a nested loop
```

```
readLoop:
    for (;;)
    {
        char[]  line;

        line = readLine();
        if (line.length > 0)
        {
            for (int i = 0;  i < line.length;  i++)
            {
                int   ch;

                ch = line[i];
                if (ch == '#')
                    break readLoop;     // Exit outer for-loop

                parse(ch);
            }
        }
        else
            return;
    }
```

Other languages, such as C and C++, do not provide a mechanism to exit more than one level of loop nesting, so goto statements must be employed.

Example N-2 - Exiting a Nested Loop

```
// [C / C++]
// Exiting a nested loop without labeled loops

    for (;;)
    {
        char  line[80];
        int   len;

        len = readLine(line);
        if (len > 0)
        {
            for (int i = 0;  i < len;  i++)
            {
                int   ch;

                ch = line[i];
                if (ch == '#')
                    goto endReadLoop;   // Exit outer for-loop

                parse(ch);
            }
        }
        else
            return;
    }
endReadLoop:;
```

This is about as clean as the Java code and just as efficient.

The alternative is to use an extra variable and extra **if** statements to avoid the use of gotos, as in Example L-2.

Example N-3 - Exiting a Nested Loop Without Goto

```
// [C / C++]
// Exiting a nested loop without goto

    bool  notDone =  true;

    while (notDone)
    {
        char  line[80];
        int   len;

        len = readLine(line);
        if (len > 0)
        {
            for (int i = 0;  notDone  &&  i < len;  i++)
            {
                int   ch;

                ch = line[i];
                if (ch == '#')
                    notDone = false;    // Exit outer while-loop
                else
                    parse(ch);
            }
        }
        else
            return;
    }
```

Exception Handling

Another common use for goto statements is in the handling of *exceptions*, or what Dijkstra called *abortion clauses*. Some languages (notably the more recent object-oriented languages) provide *exception handling* mechanisms for dealing with *synchronous* error conditions, while older languages do not.

The code below is a C function that utilizes goto statements fairly effectively for error handling and recovery. Since C does not not have any kind of exception handling mechanism, well-crafted goto statements provide a reasonable substitute.

Consider a procedure that performs four operations:

1. Allocate a control object.
2. Save a copy of a specified file name.
3. Open the named file.

4. Read a header block from the opened file.

The following example, written in C, does these operations:

Example E-1 - Error Handling Using Gotos

```
// open_control() -- [C]
// Open a file and assign it a control object.
// Returns the control object on success, or NULL on failure.

struct Control * open_control(const char *fname)
{
    struct Control *  ctl = NULL;
    FILE *            fp =  NULL;

    // 1. Allocate a control object
    ctl = malloc(sizeof(struct Control));
    memset(ctl, 0, sizeof(struct Control));
    if (ctl == NULL)                        // E-1
        goto fail;

    // 2. Save the file name
    ctl->name = malloc(strlen(fname)+1);
    if (ctl->name == NULL)                  // E-2
        goto fail;
    strcpy(ctl->name, fname);

    // 3. Open the named file
    fp = fopen(fname, "rb");
    if (fp == NULL)                         // E-3
        goto fail;
    ctl->fp = fp;

    // 4. Read the file header block
    if (!read_header(ctl))                  // E-4
        goto fail;

    // Return success
    return ctl;

fail:
    // Failure occurred, clean up allocated resources
    if (ctl != NULL)
    {
        if (ctl->fp != NULL)
            fclose(ctl->fp);                // H-3
        if (ctl->name != NULL)
            free(ctl->name);                // H-2
        free(ctl);                          // H-1
    }

    // Return failure
    return NULL;
}
```

Any of the four operations can fail, which causes the whole function to fail. After each failure, however, resources must be deallocated. Thus a failure at point E-1 requires corresponding clean-up code at point H-1, and likewise for failures at E-2 and E-3. These clean-up operations are performed in the reverse order in which their corresponding allocation operations are performed.

This type of use of goto statements is generally accepted as a "correct" use of goto. Specifically, using gotos for error handling is generally considered acceptable programming style, at least for languages (like C) that do not provide exception handling control structures.

One must be careful, however, to make it obvious that the goto statements are for error handling, e.g., by choosing an appropriately descriptive name for the goto label.

If we apply Dijkstra's maxim and remove all of the gotos, we get something like the following.

Example E-2 - Error Handling With Gotos Removed

```
// open_control() -- [C, version 2, without gotos]
// Open a file and assign it a control object.
// Returns the control object on success, or NULL on failure.

struct Control * open_control(const char *fname)
{
    struct Control *  ctl = NULL;
    FILE *            fp =  NULL;

    // 1. Allocate a control object
    ctl = malloc(sizeof(struct Control));
    memset(ctl, 0, sizeof(struct Control));
    if (ctl == NULL)                          // E-1
    {
        // Failure, clean up
        return NULL;
    }

    // 2. Save the file name
    ctl->name = malloc(strlen(fname)+1);
    if (ctl->name == NULL)                    // E-2
    {
        // Failure, clean up
        free(ctl);                            // H-1
        return NULL;
    }
    strcpy(ctl->name, fname);

    // 3. Open the named file
    fp = fopen(fname, "rb");
    if (fp == NULL)                           // E-3
    {
        // Failure, clean up
        free(ctl->name);                      // H-2
        free(ctl);                            // H-1
        return NULL;
    }
    ctl->fp = fp;
    // 4. Read the file header block
    if (!read_header(ctl))                  // E-4
    {
        // Failure, clean up
        fclose(ctl->fp);                    // H-3
        free(ctl->name);                    // H-2
        free(ctl);                          // H-1
        return NULL;
    }

    // Success
    return ctl;
}
```

The problem with this style of error handling is that we end up with a lot of duplicated clean-up code.

This suggests an alternate style which also does not use gotos but avoids code duplication.

Example E-3 - Error Handling With Gotos Removed

```
// open_control() -- [C, version 3, without gotos]
// Open a file and assign it a control object.
// Returns the control object on success, or NULL on failure.

struct Control * open_control(const char *fname)
{
    struct Control *  ctl = NULL;
    FILE *            fp =  NULL;
    int               err = 0;

    // 1. Allocate a control object
    ctl = malloc(sizeof(struct Control));
    memset(ctl, 0, sizeof(struct Control));
    if (ctl == NULL)                        // E-1
        err = 1;

    // 2. Save the file name
    if (err == 0)
    {
        ctl->name = malloc(strlen(fname)+1);
        if (ctl->name == NULL)              // E-2
            err = 2;
        else
            strcpy(ctl->name, fname);
    }

    // 3. Open the named file
    if (err == 0)
    {
        fp = fopen(fname, "rb");
        if (fp == NULL)                     // E-3
            err = 3;
        else
```

```
            ctl->fp = fp;
    }

    // 4. Read the file header block
    if (err == 0)
    {
        if (!read_header(ctl))              // E-4
            err = 4;
    }

    // Check for success
    if (err == 0)
        return ctl;

    // Failure, clean up
    if (err > 3)
        fclose(ctl->fp);                    // H-3
    if (err > 2)
        free(ctl->name);                    // H-2
    if (err > 1)
        free(ctl);                          // H-1

    return NULL;
}
```

Note that, as before, the clean-up operations are performed in the reverse order that their corresponding allocation operations are performed.

This style of error handling comes close to being as clean and succint as the style used in Example E-1. However, it requires an additional error indicator variable and extra conditional (**if**) statements.

If the function of Example E-1 is written in C++, it can take advantage of the fact that C++ supports an exception handling mechanism (i.e., **try-catch** statements), which makes the error handling in the code more obvious:

Example T-1 - Error Handling Without Gotos

```
// openControl() -- [C++]
// Open a file and assign it a control object.
// Returns the control object on success, or NULL on failure.

Control * openControl(const char *fname)
{
    Control *   ctl =   NULL;
    FILE *      fp =    NULL;

    try
    {
        // 1. Allocate a control object
        ctl = new Control;
        if (ctl == NULL)                    // E-1
            throw 1;

        // 2. Save the file name
        ctl->name = new char[::strlen(fname)+1];
```

```
            if (ctl->name == NULL)              // E-2
                throw 2;
            ::strcpy(ctl->name, fname);

            // 3. Open the named file
            fp = ::fopen(fname, "rb");
            if (fp == NULL)                     // E-3
                throw 3;
            ctl->fp = fp;

            // 4. Read the file header block
            if (not ctl->readHeader())          // E-4
                throw 4;

            // Return success
            return ctl;
        }
        catch (int err)
        {
            // Failure occurred, clean up allocated resources
            if (ctl != NULL)
            {
                if (ctl->fp != NULL)
                    ::fclose(ctl->fp);          // H-3

                if (ctl->name != NULL)
                    delete[] ctl->name;         // H-2

                delete ctl;                     // H-1
            }

            // Return failure
            return NULL;
        }
    }
```

Note that the amount of work required to clean up after a failure is exactly the same as in Example E-1. Moreover, the explicit use of a **try-catch** statement makes it obvious that the code is for error handling and recovery.

Since C++ is an object-oriented language, and thus allows the programmer to have more explicit control over object allocation and deallocation, we can make the destructor function for **Control** objects perform most of the clean-up operations. Moving the operations at H-2 and H-3 into the destructor allows us to write a simpler **catch** clause to handle failures.

Example T-2 - Error Handling Without Gotos

```
    // openControl() -- [C++, version 2]
    // Open a file and assign it a control object.
    // Returns the control object on success, or NULL on failure.

    Control * openControl(const char *fname)
    {
        Control *   ctl =   NULL;
        FILE *      fp =    NULL;
        try
        {
            ... same as Example T-1 ...
        }
        catch (int err)
        {
            // Failure occurred, clean up
            delete ctl;                             // H-1, H-2, H-3

            // Return failure
            return NULL;
        }
    }


    // Control::~Control() -- destructor

    Control::~Control()
    {
        // Clean up allocated resources
        if (this->fp != NULL)
            ::fclose(this->fp);                     // H-3
        this->fp = NULL;

        if (this->name != NULL)
            delete[] this->name;                    // H-2
        this->name = NULL;
    }
```

While the **try-catch** solution is cleaner than using **goto**s, it has the disadvantage of requiring more overhead for managing the **try** and **catch** clauses, which can be fairly expensive.

Some languages, such as Java, provide a **finally** clause as part of the **try-catch** statement to provide a way to specify actions that must be taken regardless of whether or not an exception occurs.

Example T-3 - finally clause

```
// [Java]

void write3(Resource dest, Item[] data)
{
    try
    {
        dest.acquire();
        dest.write(data[0]);
        dest.write(data[1]);
        dest.write(data[2]);
    }
    catch (ResourceException ex)
    {
        // Failure occurred, clean up
        log.error(ex);
        dest.reset();
```

```
            }
            finally
            {
                // Always executed
                dest.release();
            }
        }
```

Other languages, such as Eiffel, provide a **retry** statement as part of exception handling to allow a procedure body to be executed again after an exception is caught, presumably after some corrective actions are performed.

Conclusion

The Tao of Goto

It is obvious that loop escape and exception handler statements make for more readable and more efficient code. Of course, such flow control mechanisms are actually just fancy gotos in disguise, being implemented underneath using machine code jump instructions. However, they do not corrupt or obfuscate the program execution state. So judging them by Dijkstra's principle of being able to track program execution deterministically, they are acceptable control flow mechanisms for high-level languages.

Examples T-2 and N-1 demonstrate that Dijkstra's maxim can be achieved *provided* that the programming language provides a reasonable set of control structures that can serve in place of simple goto statements.

Examples E-1 and N-2 demonstrate the corollary to this, that if a programming language does *not* provide reasonably powerful flow control structures, there are programming problems that can be solved reasonably well *only* by resorting to the use of goto statements.

Some structured programming languages do not provide goto statements at all. Languages such as Smalltalk, Eiffel, and Java provide control statements for early and nested loop exits and exception handling, so goto is not really needed. Other languages such as Modula-2 and Oberon also do not provide goto, but appear to lack enough flow control constructs to make it convenient to write early loop exits and exception handling code; it would seem that such languages were linguistic experiments that took Dijkstra's maxim too far and failed.

Dijkstra's belief that unstructured goto statements are detrimental to good programming is still true. A properly designed language should provide flow control constructs that are powerful enough to deal with almost any programming problem. By the same token, programmers who must use languages that do not provide sufficiently flexible flow control statements should exercise restraint when using unstructured alternatives. This is the Tao of goto: knowing when to use it for good and when not to use it for evil.

In parting, I can't resist giving one last example of goto statements. I came across this code in an LR parser library I used as part of a larger compiler project (circa 1988). It is a marvelous little gem of programming succintness and simplicity.

Last Example - Nontrivial Gotos

```
int parse()
{
    Token   tok;

reading:
    tok = gettoken();
    if (tok == END)
        return ACCEPT;
shifting:
    if (shift(tok))
        goto reading;
reducing:
    if (reduce(tok))
        goto shifting;
    return ERROR;
}
```

I leave it as an exercise for the reader to rewrite this without using goto statements.

References

A. **Go To Considered Harmful**
Edsger W. Dijkstra
Letter to *Communications of the ACM* (CACM)
vol. 11 no. 3, March 1968, pp. 147-148.
Online at: www.acm.org/classics/oct95

B. Biography of **Edsger W. Dijkstra**
born May 1930, died Aug 2002
Wikipedia: en.wikipedia.org/wiki/Dijkstra

C. Biography of **C. A. R. (Tony) Hoare**
Wikipedia: en.wikipedia.org/wiki/C._A._R._Hoare

D. Biography of **Niklaus Wirth**
Wikipedia: en.wikipedia.org/wiki/Wirth
Home page: www.cs.inf.ethz.ch/~wirth

E. **Goto** programming statement
Wikipedia: en.wikipedia.org/wiki/GOTO

F. **Formal Verification**
Wikipedia: en.wikipedia.org/wiki/Formal_verification

G. **Program Derivation**
Wikipedia: en.wikipedia.org/wiki/Program_derivation

H. **Hoare Logic**
Wikipedia: en.wikipedia.org/wiki/Hoare_logic

I. **An Axiomatic Basis For Computer Programming**
C. A. R. (Tony) Hoare
Communications of the ACM (CACM)
v.12 n.10, Oct 1969
portal.acm.org/citation.cfm?doid=363235.363259 (subscription required)

J. **Programming Language Newsgroups**
groups.google.com/groups/dir?q=comp.lang.*
