Goto

GoTo (**goto**, **GOTO**, **GO TO** or other case combinations, depending on the programming language) is a statement found in many computer programming languages. It performs a **one-way transfer** of control to another line of code; in contrast a function call normally returns control. The jumped-to locations are usually identified using labels, though some languages use line numbers. At the machine code level, a `goto` is a form of branch or jump statement, in some cases combined with a stack adjustment. Many languages support the `goto` statement, and many do not (see § language support).

The structured program theorem proved that the `goto` statement is not necessary to write programs that can be expressed as flow charts; some combination of the three programming constructs of sequence, selection/choice, and repetition/iteration are sufficient for any computation that can be performed by a Turing machine, with the caveat that code duplication and additional variables may need to be introduced.[1]

In the past there was considerable debate in academia and industry on the merits of the use of `goto` statements. Use of goto was formerly common, but since the advent of structured programming in the 1960s and 1970s its use has declined significantly. The primary criticism is that code that uses goto statements is harder to understand than alternative constructions. Goto remains in use in certain common usage patterns, but alternatives are generally used if available. Debates over its (more limited) uses continue in academia and software industry circles.

Contents

- **Usage**
- **Criticism**
- **Common usage patterns**
- **Alternatives**
  - Structured programming
  - Exceptions
  - Tail calls
  - Coroutines
  - Continuations
  - Message passing
- **Variations**
  - Computed GOTO and Assigned GOTO
  - ALTER
  - Perl GOTO
  - Emulated GOTO
  - PL/I label variables
  - MS/DOS GOTO
- **Language support**
- **See also**
- **References**

Usage

goto *label*

The `goto` statement is often combined with the if statement to cause a conditional transfer of control.

IF *condition* **THEN goto** *label*

Programming languages impose different restrictions with respect to the destination of a `goto` statement. For example, the C programming language does not permit a jump to a label contained within another function,[2] however jumps within a single call chain are possible using the setjmp/longjmp functions.

Criticism

At the pre-ALGOL meeting held in 1959 Heinz Zemanek explicitly threw doubt on the necessity for GOTO statements; at the time no one paid attention to his remark, including Edsger W. Dijkstra, who later became the iconic opponent of GOTO.[3] The 1970s and 1980s saw a decline in the use of GOTO statements in favor of the "structured programming" paradigm, with goto criticized as leading to "unmaintainable spaghetti code" (see below). Some programming style coding standards, for example the GNU Pascal Coding Standards, recommend against the use of GOTO statements.[4] The Böhm–Jacopini proof (1966) did not settle the question of whether to adopt structured programming for software development, partly because the construction was more likely to obscure a program than to improve it because its application requires the introduction of additional local variables.[5] It did, however, spark a prominent debate among computer scientists, educators, language designers and application programmers that saw a slow but steady shift away from the formerly ubiquitous use of the GOTO. Probably the most famous criticism of GOTO is a 1968 letter by Edsger Dijkstra called *Go To Statement Considered Harmful*.[3][6] In that letter Dijkstra argued that unrestricted GOTO statements should be abolished from higher-level languages because they complicated the task of analyzing and verifying the correctness of programs (particularly those involving loops). The letter itself sparked a debate, including a "'GOTO Considered Harmful' Considered Harmful" letter[7] sent to *Communications of the ACM* (CACM) in March 1987, as well as further replies by other people, including Dijkstra's *On a Somewhat Disappointing Correspondence*.[8]

An alternative viewpoint is presented in Donald Knuth's *Structured Programming with go to Statements*, which analyzes many common programming tasks and finds that in some of them GOTO is the optimal language construct to use.[9] In *The C Programming Language*, Brian Kernighan and Dennis Ritchie warn that `goto` is "infinitely abusable", but also suggest that it could be used for end-of-function error handlers and for multi-level breaks from loops.[10] These two patterns can be found in numerous subsequent books on C by other authors;[11][12][13][14] a 2007 introductory textbook notes that the error handling pattern is a way to work around the "lack of built-in exception handling within the C language".[11] Other programmers, including Linux Kernel designer and coder Linus Torvalds or software engineer and book author Steve McConnell, also object to Dijkstra's point of view, stating that GOTOs can be a useful language feature, improving program speed, size and code clarity, but only when used in a sensible way by a comparably sensible programmer.[15][16] According to computer science professor John Regehr, in 2013, there were about 100,000 instances of goto in the Linux kernel code.[17]

Other academics took a more extreme viewpoint and argued that even instructions like `break` and `return` from the middle of loops are bad practice as they are not needed in the Böhm–Jacopini result, and thus advocated that loops should have a single exit point.[18] For instance, Bertrand Meyer wrote in his 2009 textbook that instructions like `break` and `continue` "are just the old `goto` in sheep's clothing".[19] A

slightly modified form of the Böhm–Jacopini result allows however the avoidance of additional variables in structured programming, as long as multi-level breaks from loops are allowed.[20] Because some languages like C don't allow multi-level breaks via their `break` keyword, some textbooks advise the programmer to use `goto` in such circumstances.[14] The MISRA C 2004 standard bans `goto`, `continue`, as well as multiple `return` and `break` statements.[21] The 2012 edition of the MISRA C standard downgraded the prohibition on `goto` from "required" to "advisory" status; the 2012 edition has an additional, mandatory rule that prohibits only backward, but not forward jumps with `goto`.[22][23]

FORTRAN introduced structured programming constructs in 1978, and in successive revisions the relatively loose semantic rules governing the allowable use of goto were tightened; the "extended range" in which a programmer could use a GOTO to enter and leave a still-executing DO loop was removed from the language in 1978,[24] and by 1995 several forms of Fortran GOTO, including the Computed GOTO and the Assigned GOTO, had been deleted.[25] Some widely used modern programming languages such as Java and Python lack the GOTO statement – see language support – though most provide some means of breaking out of a selection, or either breaking out of or moving on to the next step of an iteration. The viewpoint that disturbing the control flow in code is undesirable may be seen in the design of some programming languages, for instance Ada[26] visually emphasizes label definitions using angle brackets.

Entry 17.10 in comp.lang.c FAQ list[27] addresses the issue of GOTO use directly, stating

> Programming style, like writing style, is somewhat of an art and cannot be codified by inflexible rules, although discussions about style often seem to center exclusively around such rules. In the case of the goto statement, it has long been observed that unfettered use of goto's quickly leads to unmaintainable spaghetti code. However, a simple, unthinking ban on the goto statement does not necessarily lead immediately to beautiful programming: an unstructured programmer is just as capable of constructing a Byzantine tangle without using any goto's (perhaps substituting oddly-nested loops and Boolean control variables, instead). Many programmers adopt a moderate stance: goto's are usually to be avoided, but are acceptable in a few well-constrained situations, if necessary: as multi-level break statements, to coalesce common actions inside a switch statement, or to centralize cleanup tasks in a function with several error returns. (...) Blindly avoiding certain constructs or following rules without understanding them can lead to just as many problems as the rules were supposed to avert. Furthermore, many opinions on programming style are just that: opinions. They may be strongly argued and strongly felt, they may be backed up by solid-seeming evidence and arguments, but the opposing opinions may be just as strongly felt, supported, and argued. It's usually futile to get dragged into "style wars", because on certain issues, opponents can never seem to agree, or agree to disagree, or stop arguing.

Common usage patterns

While overall usage of gotos has been declining, there are still situations in some languages where a goto provides the shortest and most straightforward way to express a program's logic (while it is possible to express the same logic without gotos, the equivalent code will be longer and often more difficult to understand). In other languages, there are structured alternatives, notably exceptions and tail calls.

Situations in which goto is often useful include:

- To make the code more readable and easier to follow[28][29]
- To make smaller programs, and get rid of code duplication [28][29]

- Implement a finite-state machine, using a state transition table and goto to switch between states (in absence of tail call elimination), particularly in automatically generated C code.[30] For example, goto in the canonical LR parser.
- Implementing multi-level break and continue if not directly supported in the language; this is a common idiom in C.[14] Although Java reserves the goto keyword, it doesn't actually implement it. Instead, Java implements labelled break and labelled continue statements.[31] According to the Java documentation, the use of gotos for multi-level breaks was the most common (90%) use of gotos in C.[32] Java was not the first language to take this approach—forbidding goto, but providing multi-level breaks— the BLISS programming language (more precisely the BLISS-11 version thereof) preceded it in this respect.[33]:960–965
- Surrogates for single-level break or continue (retry) statements when the potential introduction of additional loops could incorrectly affect the control flow. This practice has been observed in Netbsd code.[34]
- Error handling (in absence of exceptions), particularly cleanup code such as resource deallocation.[11][14][34][30][35] C++ offers an alternative to goto statement for this use case, which is : Resource Acquisition Is Initialization (RAII) through using destructors or using try and catch exceptions used in Exception handling.[36] setjmp and longjmp are another alternative, and have the advantage of being able to unwind part of the call stack.
- popping the stack in, e.g., Algol, PL/I.

These uses are relatively common in C, but much less common in C++ or other languages with higher-level features.[35] However, throwing and catching an exception inside a function can be extraordinarily inefficient in some languages; a prime example is Objective-C, where a goto is a much faster alternative.[37]

Another use of goto statements is to modify poorly factored legacy code, where avoiding a goto would require extensive refactoring or code duplication. For example, given a large function where only certain code is of interest, a goto statement allows one to jump to or from only the relevant code, without otherwise modifying the function. This usage is considered code smell, but finds occasional use.

Alternatives

Structured programming

The modern notion of subroutine was invented by David Wheeler when programming the EDSAC.[38] To implement a call and return on a machine without a stack, he used a special pattern of self-modifying code, known as a Wheeler jump.[39] This resulted in the ability to structure programs using well-nested executions of routines drawn from a library. This would not have been possible using only `goto`, since the target code, being drawn from the library, would not know where to jump back to.

Later, high-level languages such as Pascal were designed around support for structured programming, which generalised from subroutines (also known as procedures or functions) towards further control structures such as:

- Loops using `while`, `repeat until` or `do`, and `for` statements
- `switch` a.k.a. `case` statements, a form of multiway branching

These new language mechanisms replaced equivalent flows which previously would have been written using `goto`s and `if`s. Multi-way branching replaces the "computed goto" in which the instruction to jump to is determined dynamically (conditionally).

Exceptions

In practice, a strict adherence to the basic three-structure template of structured programming yields highly nested code, due to inability to exit a structured unit prematurely, and a combinatorial explosion with quite complex program state data to handle all possible conditions.

Two solutions have been generally adopted: a way to exit a structured unit prematurely, and more generally exceptions – in both cases these go *up* the structure, returning control to enclosing blocks or functions, but do not jump to arbitrary code locations. These are analogous to the use of a return statement in non-terminal position – not strictly structured, due to early exit, but a mild relaxation of the strictures of structured programming. In C, `break` and `continue` allow one to terminate a loop or continue to the next iteration, without requiring an extra `while` or `if` statement. In some languages multi-level breaks are also possible. For handling exceptional situations, specialized exception handling constructs were added, such as `try`/`catch`/`finally` in Java.

The throw-catch exception handling mechanisms can also be easily abused to create non-transparent control structures, just like goto can be abused.[40]

Tail calls

In a paper delivered to the ACM conference in Seattle in 1977, Guy L. Steele summarized the debate over the GOTO and structured programming, and observed that procedure calls in the tail position of a procedure can be most optimally treated as a direct transfer of control to the called procedure, typically eliminating unnecessary stack manipulation operations.[41] Since such "tail calls" are very common in Lisp, a language where procedure calls are ubiquitous, this form of optimization considerably reduces the cost of a procedure call compared to the GOTO used in other languages. Steele argued that poorly implemented procedure calls had led to an artificial perception that the GOTO was cheap compared to the procedure call. Steele further argued that "in general procedure calls may be usefully thought of as GOTO statements which also pass parameters, and can be uniformly coded as machine code JUMP instructions", with the machine code stack manipulation instructions "considered an optimization (rather than vice versa!)".[41] Steele cited evidence that well optimized numerical algorithms in Lisp could execute faster than code produced by then-available commercial Fortran compilers because the cost of a procedure call in Lisp was much lower. In Scheme, a Lisp dialect developed by Steele with Gerald Jay Sussman, tail call optimization is mandatory.[42]

Although Steele's paper did not introduce much that was new to computer science, at least as it was practised at MIT, it brought to light the scope for procedure call optimization, which made the modularity-promoting qualities of procedures into a more credible alternative to the then-common coding habits of large monolithic procedures with complex internal control structures and extensive state data. In particular, the tail call optimizations discussed by Steele turned the procedure into a credible way of implementing iteration through single tail recursion (tail recursion calling the same function). Further, tail call optimization allows mutual recursion of unbounded depth, assuming tail calls – this allows transfer of control, as in finite state machines, which otherwise is generally accomplished with goto statements.

Coroutines

Coroutines are a more radical relaxation of structured programming, allowing not only multiple exit points (as in returns in non-tail position), but also multiple entry points, similar to goto statements. Coroutines are more restricted than goto, as they can only *resume* a currently running coroutine at specified points – continuing after a yield – rather than jumping to an arbitrary point in the code. A limited form of coroutines are generators, which are sufficient for some purposes. Even more limited are closures – subroutines which maintain state (via

static variables), but not execution position. A combination of state variables and structured control, notably an overall switch statement, can allow a subroutine to resume execution at an arbitrary point on subsequent calls, and is a structured alternative to goto statements in the absence of coroutines; this is a common idiom in C, for example.

Continuations

A continuation is similar to a GOTO in that it transfers control from an arbitrary point in the program to a previously marked point. A continuation is more flexible than GOTO in those languages that support it, because it can transfer control out of the current function, something that a GOTO cannot do in most structured programming languages. In those language implementations that maintain stack frames for storage of local variables and function arguments, executing a continuation involves adjusting the program's call stack in addition to a jump. The longjmp function of the C programming language is an example of an escape continuation that may be used to escape the current context to a surrounding one. The Common Lisp GO operator also has this stack unwinding property, despite the construct being lexically scoped, as the label to be jumped to can be referenced from a closure.

In Scheme, continuations can even move control from an outer context to an inner one if desired. This almost limitless control over what code is executed next makes complex control structures such as coroutines and cooperative multitasking relatively easy to write.[43]

Message passing

In non-procedural paradigms, goto is less relevant or completely absent. One of the main alternatives is message passing, which is of particular importance in concurrent computing, interprocess communication, and object oriented programming. In these cases, the individual components do not have arbitrary transfer of control, but the overall control may be scheduled in complex ways, such as via preemption. The influential languages Simula and Smalltalk were among the first to introduce the concepts of messages and objects. By encapsulating state data, object-oriented programming reduced software complexity to interactions (messages) between objects.

Variations

There are a number of different language constructs under the class of *goto* statements.

Computed GOTO and Assigned GOTO

In Fortran, a **computed `GOTO`** jumps to one of several labels in a list, based on the value of an expression. An example is `goto (20,30,40) i`. The equivalent construct in C is the switch statement and in newer Fortran a `CASE` statement is the recommend syntactical alternative.[44] BASIC has the `ON ... GOTO` construct that achieves the same goal.[45]

In versions prior to Fortran 95, Fortran also had an **assigned goto** variant that transfers control to a statement label (line number) which is stored in (assigned to) an integer variable. Jumping to an integer variable that had not been ASSIGNed to was unfortunately possible, and was a major source of bugs involving assigned gotos.[46] The Fortran `assign` statement only allows a constant (existing) line number to be assigned to the integer variable. However, it was possible to accidentally treat this variable as an integer thereafter, for example increment it, resulting in unspecified behavior at `goto` time. The following code demonstrates the behavior of the `goto i` when line *i* is unspecified:[47]

```
    assign 200 to i
    i = i+1
    goto i ! unspecified behavior
200 write(*,*) "this is valid line number"
```

Several C compilers implement two non-standard C/C++ extensions relating to gotos originally introduced by gcc.[48][49] The GNU extension allows the address of a label inside the current function to be obtained as a `void*` using the unary, prefix **label value operator** `&&`. The goto instruction is also extended to allow jumping to an arbitrary `void*` expression. This C extension is referred to as a *computed goto* in documentation of the C compilers that support it; its semantics are a superset of Fortran's assigned goto, because it allows arbitrary pointer expressions as the goto target, while Fortran's assigned goto doesn't allow arbitrary expressions as jump target.[50] As with the standard goto in C, the GNU C extension allows the target of the computed goto to reside only in the current function. Attempting to jump outside the current function results in unspecified behavior.[50]

Some variants of BASIC also support a computed GOTO in the sense used in GNU C, i.e. in which the target can be *any* line number, not just one from a list. For example, in MTS BASIC one could write `GOTO i*1000` to jump to the line numbered 1000 times the value of a variable *i* (which might represent a selected menu option, for example).[51]

PL/I *label variables* achieve the effect of computed or assigned `GOTO`s.

ALTER

Up to the 1985 ANSI COBOL standard had the ALTER verb which could be used to change the destination of an existing GO TO, which had to be in a paragraph by itself.[52] The feature, which allowed polymorphism, was frequently condemned and seldom used.[53]

Perl GOTO

In Perl, there is a variant of the `goto` statement that is not a traditional GOTO statement at all. It takes a function name and transfers control by effectively substituting one function call for another (a tail call): the new function will not return to the GOTO, but instead to the place from which the original function was called.[54]

Emulated GOTO

There are several programming languages that do not support GOTO by default. By using GOTO emulation, it is still possible to use GOTO in these programming languages, albeit with some restrictions. One can emulate GOTO in Java,[55] JavaScript,[56] and Python.[57][58]

PL/I label variables

PL/I has the data type *LABEL*, which can be used to implement both the "assigned goto" and the "computed goto." PL/I allows branches out of the current block. A calling procedure can pass a label as an argument to a called procedure which can then exit with a branch. The value of a label variable includes the address of a stack frame, and a goto out of block pops the stack.

```
  /* This implements the equivalent of */
  /* the assigned goto                 */
    declare where label;
    where = somewhere;
    goto where;
    ...
  somewhere: /* statement */ ;
    ...
```

```
  /* This implements the equivalent of */
  /* the computed goto                 */
    declare where (5) label;
    declare inx fixed;
    where(1) = abc;
    where(2) = xyz;
    ...
    goto where(inx);
    ...
  abc: /* statement */ ;
    ...
  xyz: /* statement */ ;
    ...
```

A simpler way to get an equivalent result is using a *label constant array* that doesn't even need an explicit declaration of a **LABEL** type variable:

```
  /* This implements the equivalent of */
  /* the computed goto                 */
    declare inx fixed;
    ...
    goto where(inx);
    ...
  where(1): /* statement */ ;
    ...
  where(2): /* statement */ ;
    ...
```

MS/DOS GOTO

Goto directs execution to a label that begins with a colon. The target of the Goto can be a variable.

```
@echo off
SET D8str=%date%
SET D8dow=%D8str:~0,3%

FOR %%D in (Mon Wed Fri) do if "%%D" == "%D8dow%" goto SHOP%%D
echo Today, %D8dow%, is not a shopping day.
goto end

:SHOPMon
echo buy pizza for lunch - Monday is Pizza day.
goto end

:SHOPWed
echo buy Calzone to take home - today is Wednesday.
goto end

:SHOPFri
echo buy Seltzer in case somebody wants a zero calorie drink.
:end
```

Language support

Many languages support the `goto` statement, and many do not. In Java, `goto` is a reserved word, but is unusable, although the compiled file.class generates GOTOs and LABELs.[59][60] Python does not have support for goto, although there are several joke modules that provide it.[57][58] There is no goto statement in Seed7 and hidden gotos like break- and continue-statements are also omitted.[61] In PHP there was no native support for `goto` until version 5.3 (libraries were available to emulate its functionality).[62]

The C# programming language has `goto`. However, it does not allow jumping to a label outside of the current scope, making it significantly less powerful and dangerous than the `goto` keyword in other programming languages. It also makes *case* and *default* statements labels, whose scope is the enclosing switch statement; *goto case* or *goto default* is often used as an explicit replacement for implicit fallthrough, which C# disallows.

Other languages may have their own separate keywords for explicit fallthroughs, which can be considered a version of `goto` restricted to this specific purpose. For example, Go uses the `fallthrough` keyword and doesn't allow implicit fallthrough at all, while Perl 5 uses `next` for explicit fallthrough by default, but also allows setting implicit fallthrough as default behavior for a module.

Most languages that have goto statements call it that, but in the early days of computing, other names were used. For example, in MAD the TRANSFER TO statement was used.[63] APL uses a right pointing arrow, → for goto.

C has goto, and it is commonly used in various idioms, as discussed above.

There is a `goto` (https://perldoc.perl.org/functions/goto.html) function in Perl as well.

Functional programming languages such as Scheme generally do not have goto, instead using continuations.

See also

- COMEFROM
- Control flow
- GOSUB
- Switch statement – a multiway branch (or conditional goto)
- Unstructured programming
- Considered harmful

References

1. David Anthony Watt; William Findlay (2004). *Programming language design concepts* (https://archive.org/details/programminglangu00watt_497). John Wiley & Sons. p. 228 (https://archive.org/details/programminglangu00watt_497/page/n246). ISBN 978-0-470-85320-7.
2. "The New C Standard: 6.8.6.1" (http://c0x.coding-guidelines.com/6.8.6.1.html). *c0x.coding-guidelines.com*.
3. Dijkstra 1968.
4. "GNU Pascal Coding Standards" (http://www.gnu-pascal.de/h-gpcs-en.html#Assorted-Tips). *www.gnu-pascal.de*.
5. Kenneth Louden, Lambert (2011). *Programming Languages: Principles and Practices* (https://archive.org/details/programminglangu00loud_140). Cengage Learning. p. 422 (https://archive.org/details/programminglangu00loud_140/page/n426). ISBN 978-1-111-52941-3.

6. "EWD 215: A Case against the GO TO Statement" (http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF) (PDF).
7. Frank Rubin (March 1987). " "GOTO Considered Harmful" Considered Harmful" (https://web.archive.org/web/20090320002214/http://www.ecn.purdue.edu/ParaMount/papers/rubin87goto.pdf) (PDF). *Communications of the ACM*. **30** (3): 195–196. doi:10.1145/214748.315722 (https://doi.org/10.1145%2F214748.315722). Archived from the original (http://www.ecn.purdue.edu/ParaMount/papers/rubin87goto.pdf) (PDF) on 2009-03-20.
8. Dijkstra, Edsger W. *On a Somewhat Disappointing Correspondence (EWD-1009)* (http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1009.PDF) (PDF). E.W. Dijkstra Archive. Center for American History, University of Texas at Austin. (transcription (http://www.cs.utexas.edu/users/EWD/transcriptions/EWD10xx/EWD1009.html)) (May, 1987)
9. Donald Knuth (1974). "Structured Programming with go to Statements" (https://www.cs.sjsu.edu/~mak/CS185C/KnuthStructuredProgrammingGoTo.pdf) (PDF). *Computing Surveys*. **6** (4): 261–301. CiteSeerX 10.1.1.103.6084 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.103.6084). doi:10.1145/356635.356640 (https://doi.org/10.1145%2F356635.356640).
10. Brian W. Kernighan; Dennis Ritchie (1988). *C Programming Language* (https://archive.org/details/cprogramminglang00bria) (2nd ed.). Prentice Hall. pp. 60–61 (https://archive.org/details/cprogramminglang00bria/page/60). ISBN 978-0-13-308621-8.
11. Michael A. Vine (2007). *C Programming for the Absolute Beginner*. Cengage Learning. p. 262. ISBN 978-1-59863-634-5.
12. Sandra Geisler (2011). *C All-in-One Desk Reference For Dummies*. John Wiley & Sons. pp. 217–220. ISBN 978-1-118-05424-6.
13. Stephen Prata (2013). *C Primer Plus*. Addison-Wesley. pp. 287–289. ISBN 978-0-13-343238-1.
14. Sartaj Sahni; Robert F. Cmelik; Bob Cmelik (1995). *Software Development in C* (https://books.google.com/books?id=78hu9aMNMZQC&pg=PA135). Silicon Press. p. 135. ISBN 978-0-929306-16-2.
15. "Archived copy" (https://web.archive.org/web/20100214095828/http://kerneltrap.org/node/553). Archived from the original (http://kerneltrap.org/node/553) on 2010-02-14. Retrieved 2010-01-30.
16. "Code Complete, First Edition" (http://www.stevemcconnell.com/ccgoto.htm). Stevemcconnell.com. Retrieved 2014-07-22.
17. "Use of Goto in Systems Code – Embedded in Academia" (http://blog.regehr.org/archives/894). *blog.regehr.org*.
18. Roberts, E. [1995] “Loop Exits and Structured Programming: Reopening the Debate,” ACM SIGCSE Bulletin, (27)1: 268–272.
19. Bertrand Meyer (2009). *Touch of Class: Learning to Program Well with Objects and Contracts*. Springer Science & Business Media. p. 189. ISBN 978-3-540-92144-8.
20. Dexter Kozen and Wei-Lung Dustin Tseng (2008). *The Böhm–Jacopini Theorem Is False, Propositionally* (http://www.cs.cornell.edu/~kozen/papers/bohmjacopini.pdf) (PDF). *Mpc 2008*. Lecture Notes in Computer Science. **5133**. pp. 177–192. CiteSeerX 10.1.1.218.9241 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.218.9241). doi:10.1007/978-3-540-70594-9_11 (https://doi.org/10.1007%2F978-3-540-70594-9_11). ISBN 978-3-540-70593-2.
21. "Why "continue" is considered as a C violation in MISRA C:2004?" (https://stackoverflow.com/questions/10975722/why-continue-is-considered-as-a-c-violation-in-misra-c2004). Stack Overflow. 2012-06-11. Retrieved 2014-07-22.
22. Mark Pitchford; Chris Tapp (2013-02-25). "MISRA C:2012: Plenty Of Good Reasons To Change" (http://electronicdesign.com/dev-tools/misra-c2012-plenty-good-reasons-change). Electronic Design. Retrieved 2014-07-22.
23. Tom Williams (March 2013). "Checking Rules for C: Assuring Reliability and Safety" (http://www.rtcmagazine.com/articles/print_article/102990). RTC Magazine. Retrieved 2014-07-22.

24. ANSI X3.9-1978. American National Standard – Programming Language FORTRAN. American National Standards Institute. Also known as ISO 1539-1980, informally known as FORTRAN 77
25. ISO/IEC 1539-1:1997. Information technology – Programming languages – Fortran – Part 1: Base language. Informally known as Fortran 95. There are a further two parts to this standard. Part 1 has been formally adopted by ANSI.
26. John Barnes (2006-06-30). *Programming in Ada 2005*. Addison Wesley. p. 114–115. ISBN 978-0-321-34078-8.
27. "Question 17.10" (http://c-faq.com/style/stylewars.html). C-faq.com. Retrieved 2014-07-22.
28. "Linux: Using goto In Kernel Code" (https://web.archive.org/web/20051128093253/http://kerneltrap.org/node/553/2131). 28 November 2005. Archived from the original (http://kerneltrap.org/node/553/2131) on 28 November 2005.
29. https://www.kernel.org/doc/Documentation/CodingStyle
30. Good uses of goto (https://web.archive.org/web/20110319021034/http://www.simon-cozens.org/content/good-uses-goto), Simon Cozens
31. "Branching Statements (The Java™ Tutorials > Learning the Java Language > Language Basics)" (http://docs.oracle.com/javase/tutorial/java/nutsandbolts/branch.html). Docs.oracle.com. 2012-02-28. Retrieved 2014-07-22.
32. "The Java Language Environment" (http://www.oracle.com/technetwork/java/simple-142616.html). Oracle.com. Retrieved 2014-07-22.
33. Brender, Ronald F. (2002). "The BLISS programming language: a history" (https://www.cs.tufts.edu/~nr/cs257/archive/ronald-brender/bliss.pdf) (PDF). *Software: Practice and Experience*. **32** (10): 955–981. doi:10.1002/spe.470 (https://doi.org/10.1002%2Fspe.470).
34. Diomidis Spinellis (27 May 2003). *Code Reading: The Open Source Perspective* (https://books.google.com/books?id=8lYbNfsAVT4C&pg=PA43). Addison-Wesley Professional. pp. 43–44. ISBN 978-0-672-33370-5.
35. When To Use Goto When Programming in C (http://www.cprogramming.com/tutorial/goto.html), Alex Allain
36. "Day 1 Keynote - Bjarne Stroustrup: C++11 Style | GoingNative 2012 | Channel 9" (http://channel9.msdn.com/Events/GoingNative/GoingNative-2012/Keynote-Bjarne-Stroustrup-Cpp11-Style). Channel9.msdn.com. 2012-02-02. Retrieved 2014-07-22.
37. David Chisnall (2012). *Objective-C Phrasebook* (https://archive.org/details/objectivecphrase0000chis). Addison-Wesley Professional. p. 249 (https://archive.org/details/objectivecphrase0000chis/page/249). ISBN 978-0-321-81375-6.
38. "David J. Wheeler • IEEE Computer Society" (https://www.computer.org/web/awards/pioneer-david-wheeler). *www.computer.org*.
39. Wilkes, M. V.; Wheeler, D. J.; Gill, S. (1951). *Preparation of Programs for an Electronic Digital Computer*. Addison-Wesley.
40. J. Siedersleben (2006). "Errors and Exceptions - Rights and Obligations". In Christophe Dony (ed.). *Advanced Topics in Exception Handling Techniques* (https://archive.org/details/advancedtopicsex00dony). Springer Science & Business Media. p. 277 (https://archive.org/details/advancedtopicsex00dony/page/n285). ISBN 978-3-540-37443-5.
41. Guy Lewis Steele, Jr.. "Debunking the 'Expensive Procedure Call' Myth, or, Procedure Call Implementations Considered Harmful, or, Lambda: The Ultimate GOTO". MIT AI Lab. AI Lab Memo AIM-443. October 1977.
42. R5RS Sec. 3.5, Richard Kelsey; William Clinger; Jonathan Rees; et al. (August 1998). "Revised^5 Report on the Algorithmic Language Scheme" (http://www.schemers.org/Documents/Standards/R5RS/). *Higher-Order and Symbolic Computation*. **3** (1): 7–105. doi:10.1023/A:1010051815785 (https://doi.org/10.1023%2FA%3A1010051815785).

43. "Revised^5 Report on the Algorithmic Language Scheme" (http://schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-9.html#%25_sec_6.4). *schemers.org*.
44. "Computed GOTO Statement (obsolescent)" (https://web.archive.org/web/20160526142532/http://www.lahey.com/docs/lfprohelp/F95ARComputed_GOTOStmt.htm). Lahey.com. Archived from the original (http://www.lahey.com/docs/lfprohelp/F95ARComputed_GOTOStmt.htm) on 2016-05-26. Retrieved 2014-07-22.
45. "Microsoft QuickBASIC: ON...GOSUB, ON...GOTO Statements QuickSCREEN" (http://www.qbasicnews.com/qboho/qckadvr@l804a.shtml). Microsoft. 1988. Retrieved 2008-07-03.
46. http://www.personal.psu.edu/jhm/f90/statements/goto_a.html
47. "ASSIGN - Label Assignment" (https://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/fortran-win/GUID-ADCD2825-BFEB-41FA-9B9B-1EC3D78EFCF7.htm). Software.intel.com. Retrieved 2014-07-22.
48. Computed goto (http://publib.boulder.ibm.com/infocenter/lnxpcomp/v7v91/index.jsp?topic=/com.ibm.vacpp7l.doc/language/ref/clrc08computed_goto.htm), IBM XL C/C++ compiler
49. "Intel® Composer XE 2013 SP1 Compilers Fixes List | Intel® Developer Zone" (https://software.intel.com/en-us/articles/intel-composer-xe-2013-compilers-sp1-fixes-list). Software.intel.com. 2013-08-12. Retrieved 2014-07-22.
50. "Labels as Values - Using the GNU Compiler Collection (GCC)" (https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html). Gcc.gnu.org. Retrieved 2014-07-22.
51. University of Michigan Computing Center (September 1974). *MTS, Michigan Terminal System* (https://books.google.com/books?id=_wLvAAAAMAAJ&pg=PA226). UM Libraries. p. 226. UOM:39015034770076.
52. HP COBOL II/XL Reference Manual (http://docs.hp.com/cgi-bin/doc3k/B3150090013.11820/72), "The ALTER statement is an obsolete feature of the 1985 ANSI COBOL standard."
53. Van Tassel, Dennie (July 8, 2004). "History of Labels in Programming Languages" (http://www.gavilan.edu/csis/languages/labels.html#_Toc76036283). Retrieved 4 January 2011.
54. Goto (http://perldoc.perl.org/perlsyn.html#Goto), from the perl.syn (Perl syntax) manual
55. "GOTO for Java" (https://web.archive.org/web/20120615003103/http://www.steike.com/code/useless/java-goto/). *steik*. July 6, 2009. Archived from the original (http://www.steike.com/code/useless/java-goto/) on June 15, 2012. Retrieved April 28, 2012.
56. Sexton, Alex. "The Summer of Goto | Official Home of Goto.js" (http://summerofgoto.com/). Retrieved April 28, 2012.
57. Hindle, Richie (April 1, 2004). "goto for Python" (http://entrian.com/goto/). *Entrian Solutions*. Hertford, UK: Entrian Solutions Ltd. Retrieved April 28, 2012. "The 'goto' module was an April Fool's joke, published on 1st April 2004. Yes, it works, but it's a joke nevertheless. Please don't use it in real code!"
58. snoack (September 19, 2015). "snoack/python-goto: A function decorator, that rewrites the bytecode, to enable goto in Python" (https://github.com/snoack/python-goto). Retrieved February 24, 2017.
59. "The Java Language Specification, Third Edition" (http://java.sun.com/docs/books/jls/third_edition/html/lexical.html#3.9). "The keywords const and goto are reserved, even though they are not currently used. This may allow a Java compiler to produce better error messages if these C++ keywords incorrectly appear in programs."
60. "The Java Language Specification, Third Edition" (http://java.sun.com/docs/books/jls/third_edition/html/statements.html#14.7). "Unlike C and C++, the Java programming language has no goto statement; identifier statement labels are used with break (§14.15) or continue (§14.16) statements appearing anywhere within the labeled statement."
61. "Seed7 Manual" (http://seed7.sourceforge.net/manual/intro.htm#Features_of_Seed7). Thomas Mertes. Retrieved 2019-09-19.

62. "goto - Manual" (http://php.net/manual/en/control-structures.goto.php). PHP. Retrieved 2014-07-22.
63. Bernard A. Galler, *The Language of Computers* (http://www.bitsavers.org/pdf/univOfMichigan/mad/Galler_TheLangOfComps_1962.pdf), University of Michigan, McGraw-Hill, 1962; pages 26-28, 197, 211.

- Dijkstra, Edsger W. (March 1968). "Letters to the editor: Go to statement considered harmful" (https://www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF) (PDF). *Communications of the ACM*. **11** (3): 147–148. doi:10.1145/362929.362947 (https://doi.org/10.1145%2F362929.362947).
- Ramshaw, L. (1988). "Eliminating go to's while preserving program structure". *Journal of the ACM*. **35** (4): 893–920. doi:10.1145/48014.48021 (https://doi.org/10.1145%2F48014.48021).

- https://golang.org/ref/spec#Fallthrough_statements
- http://web.engr.uky.edu/~elias/tutorials/perldoc-html/Switch.html#Allowing-fall-through

---