Control flow

In computer science, **control flow** (or **flow of control**) is the order in which individual statements, instructions or function calls of an imperative program are executed or evaluated. The emphasis on explicit control flow distinguishes an *imperative programming* language from a *declarative programming* language.

Within an imperative programming language, a *control flow statement* is a statement that results in a choice being made as to which of two or more paths to follow. For non-strict functional languages, functions and language constructs exist to achieve the same result, but they are usually not termed control flow statements.

A set of statements is in turn generally structured as a block, which in addition to grouping, also defines a lexical scope.

Interrupts and signals are low-level mechanisms that can alter the flow of control in a way similar to a subroutine, but usually occur as a response to some external stimulus or event (that can occur asynchronously), rather than execution of an *in-line* control flow statement.

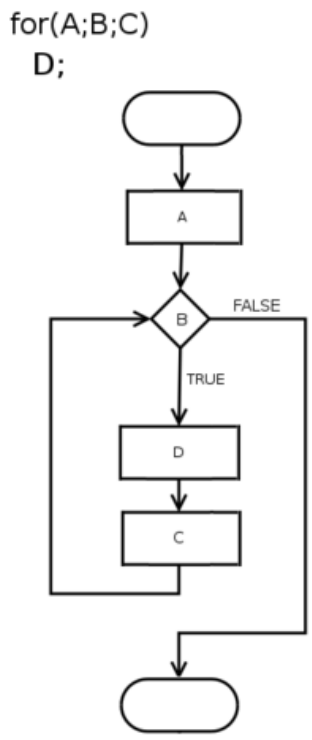
At the level of machine language or assembly language, control flow instructions usually work by altering the program counter. For some central processing units (CPUs), the only control flow instructions available are conditional or unconditional branch instructions, also termed jumps.

Contents

Categories

The kinds of control flow statements supported by different languages vary, but can be categorized by their effect:

- Continuation at a different statement (unconditional branch or jump)
- Executing a set of statements only if some condition is met (choice - i.e., conditional branch)
- Executing a set of statements zero or more times, until some condition is met (i.e., loop - the same as conditional branch)
- Executing a set of distant statements, after which the flow of control usually returns (subroutines, coroutines, and continuations)
- Stopping the program, preventing any further execution (unconditional halt)



A flow chart showing control flow.

Primitives

Labels

A label is an explicit name or number assigned to a fixed position within the source code, and which may be referenced by control flow statements appearing elsewhere in the source code. A label marks a position within source code and has no other effect.

Line numbers are an alternative to a named label used in some languages (such as BASIC). They are whole numbers placed at the start of each line of text in the source code. Languages which use these often impose the constraint that the line numbers must increase in value in each following line, but may not require that they be consecutive. For example, in BASIC:

```
10 LET X = 3
20 PRINT X
```

In other languages such as C and Ada, a label is an identifier, usually appearing at the start of a line and immediately followed by a colon. For example, in C:

```
Success: printf("The operation was successful.\n");
```

The language ALGOL 60 allowed both whole numbers and identifiers as labels (both linked by colons to the following statement), but few if any other ALGOL variants allowed whole numbers. Early Fortran compilers only allowed whole numbers as labels. Beginning with Fortran-90, alphanumeric labels have also been allowed.

Goto

The *goto* statement (a combination of the English words *go* and *to*, and pronounced accordingly) is the most basic form of unconditional transfer of control.

Although the keyword may either be in upper or lower case depending on the language, it is usually written as:

```
    goto label
```

The effect of a goto statement is to cause the next statement to be executed to be the statement appearing at (or immediately after) the indicated label.

Goto statements have been considered harmful by many computer scientists, notably Dijkstra.

Subroutines

The terminology for subroutines varies; they may alternatively be known as routines, procedures, functions (especially if they return results) or methods (especially if they belong to classes or type classes).

In the 1950s, computer memories were very small by current standards so subroutines were used mainly to reduce program size. A piece of code was written once and then used many times from various other places in a program.

Today, subroutines are more often used to help make a program more structured, e.g., by isolating some algorithm or hiding some data access method. If many programmers are working on one program, subroutines are one kind of modularity that can help divide the work.

Sequence

In structured programming, the ordered sequencing of successive commands is considered one of the basic control structures, which is used as a building block for programs alongside iteration, recursion and choice.

Minimal structured control flow

In May 1966, Böhm and Jacopini published an article[1] in *Communications of the ACM* which showed that any program with **goto**s could be transformed into a goto-free form involving only choice (IF THEN ELSE) and loops (WHILE condition DO xxx), possibly with duplicated code and/or the addition of Boolean variables (true/false flags). Later authors showed that choice can be replaced by loops (and yet more Boolean variables).

That such minimalism is possible does not mean that it is necessarily desirable; after all, computers theoretically need only one machine instruction (subtract one number from another and branch if the result is negative), but practical computers have dozens or even hundreds of machine instructions.

What Böhm and Jacopini's article showed was that all programs could be goto-free. Other research showed that control structures with one entry and one exit were much easier to understand than any other form, mainly because they could be used anywhere as a statement without disrupting the control flow. In other words, they were *composable*. (Later developments, such as non-strict programming languages – and more recently, composable software transactions – have continued this strategy, making components of programs even more freely composable.)

Some academics took a purist approach to the Böhm-Jacopini result 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 proof, and thus they advocated that all loops should have a single exit point. This purist approach is embodied in the language Pascal (designed in 1968–1969), which up to the mid-1990s was the preferred tool for teaching introductory programming in academia.[2] The direct application of the Böhm-Jacopini theorem may result in additional local variables being introduced in the structured chart, and may also result in some code duplication.[3] Pascal is affected by both of these problems and according to empirical studies cited by Eric S. Roberts, student programmers had difficulty formulating correct solutions in Pascal for several simple problems, including writing a function for searching an element in an array. A 1980 study by Henry Shapiro cited by Roberts found that using only the Pascal-provided control structures, the correct solution was given by only 20% of the subjects, while no subject wrote incorrect code for this problem if allowed to write a return from the middle of a loop.[2]

Control structures in practice

Most programming languages with control structures have an initial keyword which indicates the type of control structure involved. Languages then divide as to whether or not control structures have a final keyword.

- No final keyword: ALGOL 60, C, C++, Haskell, Java, Pascal, Perl, PHP, PL/I, Python, PowerShell. Such languages need some way of grouping statements together:
  - ALGOL 60 and Pascal: `begin` ... `end`
  - C, C++, Java, Perl, PHP, and PowerShell: curly brackets { ... }
  - PL/I: `DO` ... `END`
  - Python: uses indent level (see Off-side rule)
  - Haskell: either indent level or curly brackets can be used, and they can be freely mixed
  - Lua: uses `do` ... `end`
- Final keyword: Ada, ALGOL 68, Modula-2, Fortran 77, Mythryl, Visual Basic. The forms of the final keyword vary:
  - Ada: final keyword is `end` + *space* + initial keyword e.g., `if` ... `end if`, `loop` ... `end loop`
  - ALGOL 68, Mythryl: initial keyword spelled backwards e.g., `if` ... `fi`, `case` ... `esac`
  - Fortran 77: final keyword is `END` + initial keyword e.g., `IF` ... `ENDIF`, `DO` ... `ENDDO`
  - Modula-2: same final keyword `END` for everything

- Visual Basic: every control structure has its own keyword. `If` ... `End If`; `For` ... `Next`; `Do` ... `Loop`; `While` ... `Wend`

Choice

If-then-(else) statements

Conditional expressions and conditional constructs are features of a programming language which perform different computations or actions depending on whether a programmer-specified boolean *condition* evaluates to true or false.

- `IF..GOTO`. A form found in unstructured languages, mimicking a typical machine code instruction, would jump to (GOTO) a label or line number when the condition was met.
- `IF..THEN..(ENDIF)`. Rather than being restricted to a jump, any simple statement, or nested block, could follow the THEN key keyword. This a structured form.
- `IF..THEN..ELSE..(ENDIF)`. As above, but with a second action to be performed if the condition is false. This is one of the most common forms, with many variations. Some require a terminal `ENDIF`, others do not. C and related languages do not require a terminal keyword, or a 'then', but do require parentheses around the condition.
- Conditional statements can be and often are nested inside other conditional statements. Some languages allow `ELSE` and `IF` to be combined into `ELSEIF`, avoiding the need to have a series of `ENDIF` or other final statements at the end of a compound statement.

Pascal:

```
if a > 0 then

writeln("yes")
else

writeln("no");
```

Ada:

```
if a > 0 then

Put_Line("yes");
else

Put_Line("no");
end if;
```

C:

```
if (a > 0) {

printf("yes");
}
else {

printf("no");
}
```

Shell script:

```
if [ $a -gt 0 ];
then
      echo "yes"
else
      echo "no"
fi
```

Python:

```
if a > 0:

print("yes")
else:

print("no")
```

Lisp:

```
(princ
  (if (plusp
a)
      "yes"
      "no"))
```

Less common variations include:

- Some languages, such as Fortran, have a *three-way* or *arithmetic if*, testing whether a numeric value is positive, negative or zero.
- Some languages have a functional form of an `if` statement, for instance Lisp's `cond`.
- Some languages have an operator form of an `if` statement, such as C's ternary operator.
- Perl supplements a C-style `if` with `when` and `unless`.
- Smalltalk uses `ifTrue` and `ifFalse` messages to implement conditionals, rather than any fundamental language construct.

Case and switch statements

Switch statements (or *case statements*, or *multiway branches*) compare a given value with specified constants and take action according to the first constant to match. There is usually a provision for a default action ("else", "otherwise") to be taken if no match succeeds. Switch statements can allow compiler optimizations, such as lookup tables. In dynamic languages, the cases may not be limited to constant expressions, and might extend to pattern matching, as in the shell script example on the right, where the `*)` implements the default case as a glob matching any string. Case logic can also be implemented in functional form, as in SQL's `decode` statement.

Pascal:

```
case someChar of
  'a': actionOnA;
  'x': actionOnX;

'y','z':actionOnYandZ;
  else
actionOnNoMatch;
end;
```

Ada:

```
case someChar is
  when 'a' =>
actionOnA;
  when 'x' =>
actionOnX;
  when 'y' | 'z' =>
actionOnYandZ;
  when others =>
actionOnNoMatch;
end;
```

C:

```
switch (someChar) {
  case 'a':
actionOnA; break;
  case 'x':
actionOnX; break;
  case 'y':
  case 'z':
actionOnYandZ;
break;
  default:
actionOnNoMatch;
}
```

Shell script:

```
case $someChar
in
   a)
actionOnA ;;
   x)
actionOnX ;;
   [yz])
actionOnYandZ ;;
   *)
actionOnNoMatch
;;
esac
```

Lisp:

```
(case some-char
  ((#\a)
action-on-a)
  ((#\x)
action-on-x)
  ((#\y #\z)
action-on-y-and-z)
  (else
action-on-no-
match))
```

Loops

A loop is a sequence of statements which is specified once but which may be carried out several times in succession. The code "inside" the loop (the *body* of the loop, shown below as *xxx*) is obeyed a specified number of times, or once for each of a collection of items, or until some condition is met, or indefinitely.

In functional programming languages, such as Haskell and Scheme, loops can be expressed by using recursion or fixed point iteration rather than explicit looping constructs. Tail recursion is a special case of recursion which can be easily transformed to iteration.

Count-controlled loops

Most programming languages have constructions for repeating a loop a certain number of times. In most cases counting can go downwards instead of upwards and step sizes other than 1 can be used.

```
    FOR I = 1 TO N           | for I := 1 to N do begin
        xxx                  |     xxx
    NEXT I                   | end;
-------------------------------------------------------------
    DO I = 1,N               | for ( I=1; I<=N; ++I ) {
        xxx                  |     xxx
    END DO                   | }
```

In these examples, if N < 1 then the body of loop may execute once (with I having value 1) or not at all, depending on the programming language.

In many programming languages, only integers can be reliably used in a count-controlled loop. Floating-point numbers are represented imprecisely due to hardware constraints, so a loop such as

```
    for X := 0.1 step 0.1 to 1.0 do
```

might be repeated 9 or 10 times, depending on rounding errors and/or the hardware and/or the compiler version. Furthermore, if the increment of X occurs by repeated addition, accumulated rounding errors may mean that the value of X in each iteration can differ quite significantly from the expected sequence 0.1, 0.2, 0.3, ..., 1.0.

Condition-controlled loops

Most programming languages have constructions for repeating a loop until some condition changes. Some variations test the condition at the start of the loop; others test it at the end. If the test is at the start, the body may be skipped completely; if it is at the end, the body is always executed at least once.

```
    DO WHILE (test)          | repeat
        xxx                  |     xxx
    LOOP                     | until test;
-----------------------------------------------
    while (test) {           | do
        xxx                  |     xxx
    }                        | while (test);
```

A control break is a value change detection method used within ordinary loops to trigger processing for groups of values. Values are monitored within the loop and a change diverts program flow to the handling of the group event associated with them.

```
    DO UNTIL (End-of-File)
       IF new-zipcode <> current-zipcode
          display_tally(current-zipcode, zipcount)

          current-zipcode = new-zipcode
          zipcount = 0
       ENDIF

       zipcount++
    LOOP
```

Collection-controlled loops

Several programming languages (e.g., Ada, D, C++11, Smalltalk, PHP, Perl, Object Pascal, Java, C#, MATLAB, Visual Basic, Ruby, Python, JavaScript, Fortran 95 and later) have special constructs which allow implicit looping through all elements of an array, or all members of a set or collection.

```
someCollection do: [:eachElement |xxx].
```

```
for Item in Collection do begin xxx end;
foreach (item; myCollection) { xxx }
foreach someArray { xxx }
foreach ($someArray as $k => $v) { xxx }
Collection<String> coll; for (String s : coll) {}
foreach (string s in myStringCollection) { xxx }
someCollection | ForEach-Object { $_ }
```

```
forall ( index = first:last:step... )
```

Scala has for-expressions, which generalise collection-controlled loops, and also support other uses, such as asynchronous programming. Haskell has do-expressions and comprehensions, which together provide similar function to for-expressions in Scala.

General iteration

General iteration constructs such as C's `for` statement and Common Lisp's `do` form can be used to express any of the above sorts of loops, and others, such as looping over some number of collections in parallel. Where a more specific looping construct can be used, it is usually preferred over the general iteration construct, since it often makes the purpose of the expression clearer.

Infinite loops

Infinite loops are used to assure a program segment loops forever or until an exceptional condition arises, such as an error. For instance, an event-driven program (such as a server) should loop forever, handling events as they occur, only stopping when the process is terminated by an operator.

Infinite loops can be implemented using other control flow constructs. Most commonly, in unstructured programming this is jump back up (goto), while in structured programming this is an indefinite loop (while loop) set to never end, either by omitting the condition or explicitly setting it to true, as `while (true) ...`. Some languages have special constructs for infinite loops, typically by omitting the condition from an indefinite loop. Examples include Ada (`loop ... end loop`),[4] Fortran (`DO ... END DO`), Go (`for { ... }`), and Ruby (`loop do ... end`).

Often, an infinite loop is unintentionally created by a programming error in a condition-controlled loop, wherein the loop condition uses variables that never change within the loop.

Continuation with next iteration

Sometimes within the body of a loop there is a desire to skip the remainder of the loop body and continue with the next iteration of the loop. Some languages provide a statement such as `continue` (most languages), `skip`,[5] or `next` (Perl and Ruby), which will do this. The effect is to prematurely terminate the innermost loop body and then resume as normal with the next iteration. If the iteration is the last one in the loop, the effect is to terminate the entire loop early.

Redo current iteration

Some languages, like Perl[6] and Ruby,[7] have a `redo` statement that restarts the current iteration from the start.

Restart loop

Ruby has a `retry` statement that restarts the entire loop from the initial iteration.[8]

Early exit from loops

When using a count-controlled loop to search through a table, it might be desirable to stop searching as soon as the required item is found. Some programming languages provide a statement such as `break` (most languages), `Exit` (Visual Basic), or `last` (Perl), which effect is to terminate the current loop immediately, and transfer control to the statement immediately after that loop. Another term for early-exit loops is loop-and-a-half.

The following example is done in Ada which supports both *early exit from loops* and *loops with test in the middle*. Both features are very similar and comparing both code snippets will show the difference: *early exit* must be combined with an **if** statement while a *condition in the middle* is a self-contained construct.

```
with Ada.Text IO;
with Ada.Integer Text IO;

procedure Print_Squares is
    X : Integer;
begin
    Read_Data : loop
        Ada.Integer Text IO.Get(X);
    exit Read_Data when X = 0;
        Ada.Text IO.Put (X * X);
        Ada.Text IO.New_Line;
    end loop Read_Data;
end Print_Squares;
```

Python supports conditional execution of code depending on whether a loop was exited early (with a `break` statement) or not by using an else-clause with the loop. For example,

```
for n in set_of_numbers:
    if isprime(n):
        print("Set contains a prime number")
        break
else:
    print("Set did not contain any prime numbers")
```

The `else` clause in the above example is linked to the `for` statement, and not the inner `if` statement. Both Python's `for` and `while` loops support such an else clause, which is executed only if early exit of the loop has not occurred.

Some languages support breaking out of nested loops; in theory circles, these are called multi-level breaks. One common use example is searching a multi-dimensional table. This can be done either via multilevel breaks (break out of *N* levels), as in bash[9] and PHP,[10] or via labeled breaks (break out and continue at given label), as in Java and Perl.[11] Alternatives to multilevel breaks include single breaks, together with a state variable which is tested to break out another level; exceptions, which are caught at the level being broken out to; placing the nested loops in a function and using return to effect termination of the entire nested loop; or using a label and a goto statement. C does not include a multilevel break, and the usual alternative is to use a goto to implement a labeled break.[12] Python does not have a multilevel break or continue – this was proposed in PEP 3136 (https://www.python.org/dev/peps/pep-3136/), and rejected on the basis that the added complexity was not worth the rare legitimate use.[13]

The notion of multi-level breaks is of some interest in theoretical computer science, because it gives rise to what is today called the *Kosaraju hierarchy*.[14] In 1973 S. Rao Kosaraju refined the structured program theorem by proving that it is possible to avoid adding additional variables in structured programming, as long as arbitrary-depth, multi-level breaks from loops are allowed.[15] Furthermore, Kosaraju proved that a strict hierarchy of programs exists: for every integer *n*, there exists a program containing a multi-level break of depth *n* that cannot be rewritten as a program with multi-level breaks of depth less than *n* without introducing added variables.[14]

One can also `return` out of a subroutine executing the looped statements, breaking out of both the nested loop and the subroutine. There are other proposed control structures for multiple breaks, but these are generally implemented as exceptions instead.

In his 2004 textbook, David Watt uses Tennent's notion of sequencer to explain the similarity between multi-level breaks and return statements. Watt notes that a class of sequencers known as *escape sequencers*, defined as "sequencer that terminates execution of a textually enclosing command or procedure", encompasses both breaks from loops (including multi-level breaks) and return statements. As commonly implemented, however, return sequencers may also carry a (return) value, whereas the break sequencer as implemented in contemporary languages usually cannot.[16]

Loop variants and invariants

Loop variants and loop invariants are used to express correctness of loops.[17]

In practical terms, a loop variant is an integer expression which has an initial non-negative value. The variant's value must decrease during each loop iteration but must never become negative during the correct execution of the loop. Loop variants are used to guarantee that loops will terminate.

A loop invariant is an assertion which must be true before the first loop iteration and remain true after each iteration. This implies that when a loop terminates correctly, both the exit condition and the loop invariant are satisfied. Loop invariants are used to monitor specific properties of a loop during successive iterations.

Some programming languages, such as Eiffel contain native support for loop variants and invariants. In other cases, support is an add-on, such as the Java Modeling Language's specification for loop statements (http://www.eecs.ucf.edu/~leavens/JML//jmlrefman/jmlrefman_12.html#SEC168) in Java.

Loop sublanguage

Some Lisp dialects provide an extensive sublanguage for describing Loops. An early example can be found in Conversional Lisp of Interlisp. Common Lisp[18] provides a Loop macro which implements such a sublanguage.

Loop system cross-reference table

Programming language	conditional			loop				early exit	loop continuation	redo	retry	correctness facilities	
	begin	middle	end	count	collection	general	infinite [1]					variant	invariant
Ada	Yes	Yes	Yes	Yes	arrays	No	Yes	deep nested	No				
APL	Yes	No	Yes	Yes	Yes	Yes	Yes	deep nested [3]	Yes	No	No		
C	Yes	No	Yes	No [2]	No	Yes	No	deep nested [3]	deep nested [3]	No			
C++	Yes	No	Yes	No [2]	Yes [9]	Yes	No	deep nested [3]	deep nested [3]	No			
C#	Yes	No	Yes	No [2]	Yes	Yes	No	deep nested [3]	deep nested [3]				
COBOL	Yes	No	Yes	Yes	No	Yes	No	deep nested [15]	deep nested [14]	No			
Common Lisp	Yes	Yes	Yes	Yes	builtin only [16]	Yes	Yes	deep nested	No				
D	Yes	No	Yes	Yes	Yes	Yes	Yes[14]	deep nested	deep nested	No			
Eiffel	Yes	No	No	Yes [10]	Yes	Yes	No	one level [10]	No	No	No [11]	integer only [13]	Yes
F#	Yes	No	No	Yes	Yes	No	No	No [6]	No	No			
FORTRAN 77	Yes	No	No	Yes	No	No	No	one level	Yes				
Fortran 90	Yes	No	No	Yes	No	No	Yes	deep nested	Yes				
Fortran 95 and later	Yes	No	No	Yes	arrays	No	Yes	deep nested	Yes				
Haskell	No	No	No	No	Yes	No	Yes	No [6]	No	No			
Java	Yes	No	Yes	No [2]	Yes	Yes	No	deep nested	deep nested	No		non-native [12]	non-native [12]
JavaScript	Yes	No	Yes	No [2]	Yes	Yes	No	deep nested	deep nested	No			
Natural	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	No		
OCaml	Yes	No	No	Yes	arrays,lists	No	No	No [6]	No	No			
PHP	Yes	No	Yes	No [2] [5]	Yes [4]	Yes	No	deep nested	deep nested	No			
Perl	Yes	No	Yes	No [2] [5]	Yes	Yes	No	deep nested	deep nested	Yes			
Python	Yes	No	No	No [5]	Yes	No	No	deep nested [6]	deep nested [6]	No			
REBOL	No [7]	Yes	Yes	Yes	Yes	No [8]	Yes	one level [6]	No	No			
Ruby	Yes	No	Yes	Yes	Yes	No	Yes	deep nested [6]	deep nested [6]	Yes	Yes		
Standard ML	Yes	No	No	No	arrays,lists	No	No	No [6]	No	No			
Visual Basic .NET	Yes	No	Yes	Yes	Yes	No	Yes	one level per type of loop	one level per type of loop				

PowerShell	Yes	No	Yes	No [2]	Yes	Yes	No	?	Yes				

1. [a] `while (true)` does not count as an infinite loop for this purpose, because it is not a dedicated language structure.
2. [a] [b] [c] [d] [e] [f] [g] [h] C's `for (init; test; increment)` loop is a general loop construct, not specifically a counting one, although it is often used for that.
3. [a] [b] [c] Deep breaks may be accomplished in APL, C, C++ and C# through the use of labels and gotos.
4. [a] Iteration over objects was added (http://www.php.net/manual/en/language.oop5.iterations.php) in PHP 5.
5. [a] [b] [c] A counting loop can be simulated by iterating over an incrementing list or generator, for instance, Python's `range()`.
6. [a] [b] [c] [d] [e] Deep breaks may be accomplished through the use of exception handling.
7. [a] There is no special construct, since the `while` function can be used for this.
8. [a] There is no special construct, but users can define general loop functions.
9. [a] The C++11 standard introduced the range-based for. In the STL, there is a `std::for_each` template function which can iterate on STL containers and call a unary function for each element.[19] The functionality also can be constructed as macro on these containers.[20]
10. [a] Count-controlled looping is effected by iteration across an integer interval; early exit by including an additional condition for exit.
11. [a] Eiffel supports a reserved word `retry`, however it is used in exception handling, not loop control.
12. [a] Requires Java Modeling Language (JML) behavioral interface specification language.
13. [a] Requires loop variants to be integers; transfinite variants are not supported. [1] (http://archive.eiffel.com/doc/faq/variant.html)
14. [a] D supports infinite collections, and the ability to iterate over those collections. This does not require any special construct.
15. [a] Deep breaks can be achieved using `GO TO` and procedures.
16. [a] Common Lisp predates the concept of generic collection type.

Structured non-local control flow

Many programming languages, especially those favoring more dynamic styles of programming, offer constructs for *non-local control flow*. These cause the flow of execution to jump out of a given context and resume at some predeclared point. *Conditions*, *exceptions* and *continuations* are three common sorts of non-local control constructs; more exotic ones also exist, such as generators, coroutines and the async keyword.

Conditions

PL/I has some 22 standard conditions (e.g., ZERODIVIDE SUBSCRIPTRANGE ENDFILE) which can be raised and which can be intercepted by: ON *condition* action; Programmers can also define and use their own named conditions.

Like the *unstructured if*, only one statement can be specified so in many cases a GOTO is needed to decide where flow of control should resume.

Unfortunately, some implementations had a substantial overhead in both space and time (especially SUBSCRIPTRANGE), so many programmers tried to avoid using conditions.

Common Syntax examples:

```
ON condition GOTO label
```

Exceptions

Modern languages have a specialized structured construct for exception handling which does not rely on the use of `GOTO` or (multi-level) breaks or returns. For example, in C++ one can write:

```
try {
    xxx1                                // Somewhere in here
    xxx2                                //     use: '''throw''' someValue;
    xxx3
} catch (someClass& someId) {           // catch value of someClass
    actionForSomeClass
} catch (someType& anotherId) {         // catch value of someType
    actionForSomeType
} catch (...) {                         // catch anything not already caught
```

```
    actionForAnythingElse
}
```

Any number and variety of `catch` clauses can be used above. If there is no `catch` matching a particular `throw`, control percolates back through subroutine calls and/or nested blocks until a matching `catch` is found or until the end of the main program is reached, at which point the program is forcibly stopped with a suitable error message.

Via C++'s influence, `catch` is the keyword reserved for declaring a pattern-matching exception handler in other languages popular today, like Java or C#. Some other languages like Ada use the keyword `exception` to introduce an exception handler and then may even employ a different keyword (`when` in Ada) for the pattern matching. A few languages like AppleScript incorporate placeholders in the exception handler syntax to automatically extract several pieces of information when the exception occurs. This approach is exemplified below by the `on error` construct from AppleScript:

```
try
    set myNumber to myNumber / 0
on error e  number n  from f  to t  partial result pr
    if ( e = "Can't divide by zero" ) then display dialog "You must not do that"
end try
```

David Watt's 2004 textbook also analyzes exception handling in the framework of sequencers (introduced in this article in the section on early exits from loops). Watt notes that an abnormal situation, generally exemplified with arithmetic overflows or input/output failures like file not found, is a kind of error that "is detected in some low-level program unit, but [for which] a handler is more naturally located in a high-level program unit". For example, a program might contain several calls to read files, but the action to perform when a file is not found depends on the meaning (purpose) of the file in question to the program and thus a handling routine for this abnormal situation cannot be located in low-level system code. Watts further notes that introducing status flags testing in the caller, as single-exit structured programming or even (multi-exit) return sequencers would entail, results in a situation where "the application code tends to get cluttered by tests of status flags" and that "the programmer might forgetfully or lazily omit to test a status flag. In fact, abnormal situations represented by status flags are by default ignored!" Watt notes that in contrast to status flags testing, exceptions have the opposite default behavior, causing the program to terminate unless the programmer explicitly deals with the exception in some way, possibly by adding explicit code to ignore it. Based on these arguments, Watt concludes that jump sequencers or escape sequencers aren't as suitable as a dedicated exception sequencer with the semantics discussed above.[21]

In Object Pascal, D, Java, C#, and Python a `finally` clause can be added to the `try` construct. No matter how control leaves the `try` the code inside the `finally` clause is guaranteed to execute. This is useful when writing code that must relinquish an expensive resource (such as an opened file or a database connection) when finished processing:

```
FileStream stm = null;                  // C# example
try
{
    stm = new FileStream("logfile.txt", FileMode.Create);
    return ProcessStuff(stm);           // may throw an exception
}
finally
{
    if (stm != null)
        stm.Close();
}
```

Since this pattern is fairly common, C# has a special syntax:

```
using (var stm = new FileStream("logfile.txt", FileMode.Create))
{
    return ProcessStuff(stm); // may throw an exception
}
```

Upon leaving the `using`-block, the compiler guarantees that the `stm` object is released, effectively binding the variable to the file stream while abstracting from the side effects of initializing and releasing the file. Python's `with` statement and Ruby's block argument to `File.open` are used to similar effect.

All the languages mentioned above define standard exceptions and the circumstances under which they are thrown. Users can throw exceptions of their own; in fact C++ allows users to throw and catch almost any type, including basic types like `int`, whereas other languages like Java aren't as permissive.

Continuations

Async

C# 5.0 introduced the async keyword for supporting asynchronous I/O in a "direct style".

Generators

Generators, also known as semicoroutines, allow control to be yielded to a consumer method temporarily, typically using a `yield` keyword (yield description (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/yield)) . Like the async keyword, this supports programming in a "direct style".

Coroutines

Coroutines are functions that can yield control to each other - a form of co-operative multitasking without threads.

Coroutines can be implemented as a library if the programming language provides either continuations or generators - so the distinction between coroutines and generators in practice is a technical detail.

Non-local control flow cross reference

Programming language	conditions	exceptions	generators/coroutines	async
Ada	No	Yes	?	?
C	No	No	No	No
C++	No	Yes	yes, by using BOOST	?
C#	No	Yes	Yes	Yes
COBOL	Yes	Yes	No	No
Common Lisp	Yes	No	?	?
D	No	Yes	Yes	?
Eiffel	No	Yes	?	?
Erlang	No	Yes	Yes	?
F#	No	Yes	Yes	Yes
Go	No	Yes	Yes	?
Haskell	No	Yes	Yes	No
Java	No	Yes	No	No
JavaScript	?	Yes	Yes, ES6	Yes, Stage 3
Objective-C	No	Yes	No	?
PHP	No	Yes	Yes	?
PL/I	Yes	No	No	No
Python	No	Yes	Yes	Yes[22]
REBOL	Yes	Yes	No	?
Ruby	No	Yes	Yes	?
Rust	No	Yes	experimental [23][24]	Yes[25]
Scala	No	Yes	via experimental extension[26]	via experimental extension
Tcl	via traces	Yes	Yes	via event loop
Visual Basic .NET	Yes	Yes	No	?
PowerShell	No	Yes	No	?

Proposed control structures

In a spoof Datamation article[27] in 1973, R. Lawrence Clark suggested that the GOTO statement could be replaced by the COMEFROM statement, and provides some entertaining examples. COMEFROM was implemented in one esoteric programming language named INTERCAL.

Donald Knuth's 1974 article "Structured Programming with go to Statements",[28] identifies two situations which were not covered by the control structures listed above, and gave examples of control structures which could handle these situations. Despite their utility, these constructs have not yet found their way into mainstream programming languages.

Loop with test in the middle

The following was proposed by Dahl in 1972:[29]

```
    loop                            loop
        xxx1                            read(char);
    while test;                     while not atEndOfFile;
        xxx2                            write(char);
    repeat;                         repeat;
```

If *xxx1* is omitted, we get a loop with the test at the top (a traditional **while** loop). If *xxx2* is omitted, we get a loop with the test at the bottom, equivalent to a **do while** loop in many languages. If **while** is omitted, we get an infinite loop. The construction here can be thought of as a **do** loop with the while check in the middle. Hence this single construction can replace several constructions in most programming languages.

Languages lacking this construct generally emulate it using an equivalent infinite-loop-with-break idiom:

```
while (true) {
    xxx1
    if (not test)
        break
    xxx2
}
```

A possible variant is to allow more than one **while** test; within the loop, but the use of **exitwhen** (see next section) appears to cover this case better.

In Ada, the above loop construct (**loop-while-repeat**) can be represented using a standard infinite loop (**loop - end loop**) that has an **exit when** clause in the middle (not to be confused with the **exitwhen** statement in the following section).

```
with Ada.Text_IO;
with Ada.Integer_Text_IO;

procedure Print_Squares is
    X : Integer;
begin
    Read_Data : loop
        Ada.Integer_Text_IO.Get(X);
    exit Read_Data when X = 0;
        Ada.Text IO.Put (X * X);
        Ada.Text IO.New_Line;
    end loop Read_Data;
end Print_Squares;
```

Naming a loop (like *Read_Data* in this example) is optional but permits leaving the outer loop of several nested loops.

Multiple early exit/exit from nested loops

This was proposed by Zahn in 1974.[30] A modified version is presented here.

```
    exitwhen EventA or EventB or EventC;
        xxx
    exits
        EventA: actionA
        EventB: actionB
        EventC: actionC
    endexit;
```

exitwhen is used to specify the events which may occur within *xxx*, their occurrence is indicated by using the name of the event as a statement. When some event does occur, the relevant action is carried out, and then control passes just after **endexit**. This construction provides a very clear separation between determining that some situation applies, and the action to be taken for that situation.

exitwhen is conceptually similar to exception handling, and exceptions or similar constructs are used for this purpose in many languages.

The following simple example involves searching a two-dimensional table for a particular item.

```
    exitwhen found or missing;
        for I := 1 to N do
            for J := 1 to M do
                if table[I,J] = target then found;
        missing;
    exits
        found:   print ("item is in table");
```

```
        missing: print ("item is not in table");
    endexit;
```

Security

One way to attack a piece of software is to redirect the flow of execution of a program. A variety of control-flow integrity techniques, including stack canaries, buffer overflow protection, shadow stacks, and vtable pointer verification, are used to defend against these attacks.[31][32][33]

See also

- Branch (computer science)
- Control-flow analysis
- Control-flow diagram
- Control-flow graph
- Control table
- Coroutine
- Cyclomatic complexity
- Drakon-chart
- Flowchart
- GOTO
- Jeroo, helps learn control structures
- Main loop
- Recursion
- Scheduling (computing)
- Spaghetti code
- Structured programming
- Subroutine
- Switch statement, alters control flow conditionally

References

1. Böhm, Jacopini. "Flow diagrams, turing machines and languages with only two formation rules" Comm. ACM, 9(5):366-371, May 1966.
2. Roberts, E. [1995] "Loop Exits and Structured Programming: Reopening the Debate (http://cs.stanford.edu/people/eroberts/papers/SIGCSE-1995/LoopExits.pdf)," ACM SIGCSE Bulletin, (27)1: 268–272.
3. David Anthony Watt; William Findlay (2004). *Programming language design concepts*. John Wiley & Sons. p. 228. ISBN 978-0-470-85320-7.
4. Ada Programming: Control: Endless Loop
5. "What is a loop and how we can use them?" (http://www.megacpptutorials.com/2012/12/what-is-loop.html). Retrieved 2020-05-25.
6. "redo - perldoc.perl.org" (https://perldoc.perl.org/functions/redo.html). *perldoc.perl.org*. Retrieved 2020-09-25.
7. "control_expressions - Documentation for Ruby 2.4.0" (https://docs.ruby-lang.org/en/2.4.0/syntax/control_expressions_rdoc.html). *docs.ruby-lang.org*. Retrieved 2020-09-25.
8. "control_expressions - Documentation for Ruby 2.3.0" (https://docs.ruby-lang.org/en/2.3.0/syntax/control_expressions_rdoc.html). *docs.ruby-lang.org*. Retrieved 2020-09-25.
9. Advanced Bash Scripting Guide: 11.3. Loop Control (http://tldp.org/LDP/abs/html/loopcontrol.html)
10. PHP Manual: "break (http://php.net/manual/en/control-structures.break.php)"
11. perldoc: last (http://perldoc.perl.org/functions/last.html)
12. comp.lang.c FAQ list · "Question 20.20b (http://c-faq.com/misc/multibreak.html)"
13. [Python-3000] Announcing PEP 3136 (http://mail.python.org/pipermail/python-3000/2007-July/008663.html), Guido van Rossum
14. Kozen, Dexter (2008). "The Böhm–Jacopini Theorem Is False, Propositionally". *Mathematics of Program Construction* (http://www.cs.cornell.edu/~kozen/papers/BohmJacopini.pdf) (PDF). 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.
15. Kosaraju, S. Rao. "Analysis of structured programs," Proc. Fifth Annual ACM Syrup. Theory of Computing, (May 1973), 240-252; also in J. Computer and System Sciences, 9, 3 (December 1974). cited by Donald Knuth (1974). "Structured Programming with go to Statements". *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). S2CID 207630080 (https://api.semanticscholar.org/CorpusID:207630080).

16. David Anthony Watt; William Findlay (2004). *Programming language design concepts*. John Wiley & Sons. pp. 215–221. ISBN 978-0-470-85320-7.
17. Meyer, Bertrand (1991). *Eiffel: The Language*. Prentice Hall. pp. 129–131.
18. "Common Lisp LOOP macro" (http://www.lispworks.com/documentation/HyperSpec/Body/m_loop.htm).
19. for_each (http://www.sgi.com/tech/stl/for_each.html). Sgi.com. Retrieved on 2010-11-09.
20. Chapter 1. Boost.Foreach (http://boost-sandbox.sourceforge.net/libs/foreach/doc/html/). Boost-sandbox.sourceforge.net (2009-12-19). Retrieved on 2010-11-09.
21. David Anthony Watt; William Findlay (2004). *Programming language design concepts*. John Wiley & Sons. pp. 221–222. ISBN 978-0-470-85320-7.
22. https://docs.python.org/3/library/asyncio.html
23. https://doc.rust-lang.org/beta/unstable-book/language-features/generators.html
24. https://docs.rs/corona/0.4.3/corona/
25. https://rust-lang.github.io/async-book/
26. http://storm-enroute.com/coroutines/
27. We don't know where to GOTO if we don't know where we've COME FROM. This (spoof) linguistic innovation lives up to all expectations. (http://www.fortran.com/fortran/come_from.html) Archived (https://web.archive.org/web/20180716171336/http://www.fortran.com/fortran/come_from.html) 2018-07-16 at the Wayback Machine By R. Lawrence Clark* From Datamation, December, 1973
28. Knuth, Donald E. "Structured Programming with go to Statements" ACM Computing Surveys 6(4):261-301, December 1974.
29. Dahl & Dijkstra & Hoare, "Structured Programming" Academic Press, 1972.
30. Zahn, C. T. "A control statement for natural top-down structured programming" presented at Symposium on Programming Languages, Paris, 1974.
31. Payer, Mathias; Kuznetsov, Volodymyr. "On differences between the CFI, CPS, and CPI properties" (https://nebelwelt.net/blog/20141007-CFICPSCPIdiffs.html). *nebelwelt.net*. Retrieved 2016-06-01.
32. "Adobe Flash Bug Discovery Leads To New Attack Mitigation Method" (http://www.darkreading.com/vulnerabilities---threats/adobe-flash-bug-discovery-leads-to-new-attack-mitigation-method/d/d-id/1323092). *Dark Reading*. Retrieved 2016-06-01.
33. Endgame. "Endgame to Present at Black Hat USA 2016" (http://www.prnewswire.com/news-releases/endgame-to-present-at-black-hat-usa-2016-300267060.html). *www.prnewswire.com*. Retrieved 2016-06-01.

Further reading

- Hoare, C. A. R. "Partition: Algorithm 63," "Quicksort: Algorithm 64," and "Find: Algorithm 65." Comm. ACM 4, 321-322, 1961.

External links

- Media related to Control flow at Wikimedia Commons
- Go To Statement Considered Harmful (https://web.archive.org/web/20070703050443/http://www.acm.org/classics/oct95/)
- A Linguistic Contribution of GOTO-less Programming (https://web.archive.org/web/20180716171336/http://www.fortran.com/fortran/come_from.html)
- "Structured Programming with Go To Statements" (https://web.archive.org/web/20090824073244/http://pplab.snu.ac.kr/courses/adv_pl05/papers/p261-knuth.pdf) (PDF). Archived from the original (http://pplab.snu.ac.kr/courses/adv_pl05/papers/p261-knuth.pdf) (PDF) on 2009-08-24. (2.88 MB)
- "IBM 704 Manual" (http://www.bitsavers.org/pdf/ibm/704/24-6661-2_704_Manual_1955.pdf) (PDF). (31.4 MB)