WIKIPEDIA

Block (programming)

In computer programming, a **block** or **code block** is a lexical structure of source code which is grouped together. Blocks consist of one or more declarations and statements. A programming language that permits the creation of blocks, including blocks nested within other blocks, is called a **block-structured programming language**. Blocks are fundamental to structured programming, where control structures are formed from blocks.

The function of blocks in programming is to enable groups of statements to be treated as if they were one statement, and to narrow the lexical scope of variables, procedures and functions declared in a block so that they do not conflict with variables having the same name used elsewhere in a program for different purposes. In a block-structured programming language, the names of variables and other objects such as procedures which are declared in outer blocks are visible inside other inner blocks, unless they are shadowed by an object of the same name.

Contents

History

Ideas of block structure were developed in the 1950s during the development of the first autocodes, and were formalized in the Algol 58 and Algol 60 reports. Algol 58 introduced the notion of the "compound statement", which was related solely to control flow.[1] The subsequent *Revised Report* which described the syntax and semantics of Algol 60 introduced the notion of a block and block scope, with a block consisting of " A sequence of declarations followed by a sequence of statements and enclosed between begin and end..." in which "[e]very declaration appears in a block in this way and is valid only for that block."[2]

Syntax

Blocks use different syntax in different languages. Two broad families are:

- the ALGOL family in which blocks are delimited by the keywords "`begin`" and "`end`"
- the C family in which blocks are delimited by curly braces - "`{`" and "`}`"

Some other techniques used are as follows :

- parentheses - "`(`" and "`)`", as in batch language and ALGOL 68.
- indentation, as in Python

- s-expressions with a syntactic keyword such as `lambda` or `let` (as in the Lisp family)
- In 1968 (with ALGOL 68), then in Edsger W. Dijkstra's 1974 Guarded Command Language the conditional and iterative code block are alternatively terminated with the block reserved word *reversed*: e.g. **`if`** ~ `then` ~ `elif` ~ `else` ~ **`fi`**, **`case`** ~ `in` ~ `out` ~ **`esac`** and `for` ~ `while` ~ **`do`** ~ **`od`**

Limitations

Some languages which support blocks with variable declarations do not fully support all declarations; for instance many C-derived languages do not permit a function definition within a block (nested functions). And unlike its ancestor Algol, Pascal does not support the use of blocks with their own declarations inside the begin and end of an existing block, only compound statements enabling sequences of statements to be grouped together in **if**, **while**, **repeat** and other control statements.

Basic semantics

The semantic meaning of a block is twofold. Firstly, it provides the programmer with a way for creating arbitrarily large and complex structures that can be treated as units. Secondly, it enables the programmer to limit the scope of variables and sometimes other objects that have been declared.

In primitive languages such as early Fortran and BASIC, there were a few built-in statement types, and little or no means of extending them in a structured manner. For instance, until 1978 standard Fortran had no "block if" statement, so to write a standard-complying code to implement simple decisions the programmer had to resort to gotos:

```
C     LANGUAGE: ANSI STANDARD FORTRAN 66
C     INITIALIZE VALUES TO BE CALCULATED
      PAYSTX = .FALSE.
      PAYSST = .FALSE.
      TAX = 0.0
      SUPTAX = 0.0
C     SKIP TAX DEDUCTION IF EMPLOYEE EARNS LESS THAN TAX THRESHOLD
      IF (WAGES .LE. TAXTHR) GOTO 100
      PAYSTX = .TRUE.
      TAX = (WAGES - TAXTHR) * BASCRT
C     SKIP SUPERTAX DEDUCTION IF EMPLOYEE EARNS LESS THAN SUPERTAX THRESHOLD
      IF (WAGES .LE. SUPTHR) GOTO 100
      PAYSST = .TRUE.
      SUPTAX = (WAGES - SUPTHR) * SUPRAT
  100 TAXED = WAGES - TAX - SUPTAX
```

Even in this very brief Fortran fragment, written to the Fortran 66 standard, it is not easy to see the structure of the program, because that structure is not reflected in the language. Without careful study it is not easy to see the circumstances in which a given statement is executed.

Blocks allow the programmer to treat a group of statements as a unit, and the default values which had to appear in initialization in this style of programming can, with a block structure, be placed closer to the decision:

```
    { Language: Jensen and Wirth Pascal }
    if wages > tax_threshold then
        begin
        paystax := true;
        tax := (wages - tax_threshold) * tax_rate
        { The block structure makes it easier to see how the code could
          be refactored for clarity, and also makes it easier to do,
          because the structure of the inner conditional can easily be moved
          out of the outer conditional altogether and the effects of doing
          so are easily predicted. }
        if wages > supertax_threshold then
            begin
            pays_supertax := true;
            supertax := (wages - supertax_threshold) * supertax_rate
            end
        else begin
            pays_supertax := false;
            supertax := 0
            end
        end
    else begin
        paystax := false; pays_supertax := false;
        tax := 0; supertax := 0
        end;
    taxed := wages - tax - supertax;
```

Use of blocks in the above fragment of Pascal enables the programmer to be clearer about what he or she intends, and to combine the resulting blocks into a nested hierarchy of conditional statements. The structure of the code reflects the programmer's thinking more closely, making it easier to understand and modify.

From looking at the above code the programmer can easily see that he or she can make the source code even clearer by taking the inner if statement out of the outer one altogether, placing the two blocks one after the other to be executed consecutively. Semantically there is little difference in this case, and the use of block structure, supported by indenting for readability, makes it easy for the programmer to refactor the code.

In primitive languages, variables had broad scope. For instance, an integer variable called IEMPNO might be used in one part of a Fortran subroutine to denote an employee social security number (ssn), but during maintenance work on the same subroutine, a programmer might accidentally use the same variable, IEMPNO, for a different purpose, and this could result in a bug that was difficult to trace. Block structure makes it easier for programmers to control scope to a minute level.

```scheme
;; Language: R5RS Standard Scheme
(let ((empno (ssn-of employee-name)))
  (while (is-manager empno)
    (let ((employees (length (underlings-of empno))))
      (printf "~a has ~a employees working under him:~%" employee-name employees)
      (for-each
        (lambda(empno)
          ;; Within this lambda expression the variable empno refers to the ssn
          ;; of an underling. The variable empno in the outer expression,
          ;; referring to the manager's ssn, is shadowed.
          (printf "Name: ~a, role: ~a~%"
                  (name-of empno)
                  (role-of empno)))
        (underlings-of empno)))))
```

In the above Scheme fragment, empno is used to identify both the manager and his or her underlings each by their respective ssn, but because the underling ssn is declared within an inner block it does not interact with the variable of the same name that contains the manager's ssn. In practice, considerations of clarity would probably lead the programmer to choose distinct variable names, but he or she has the choice and it is more difficult to introduce a bug inadvertently.

Hoisting

In a few circumstances, code in a block is evaluated as if the code were actually at the top of the block or outside the block. This is often colloquially known as *hoisting*, and includes:

- Loop-invariant code motion, a compiler optimization where code in the loop that is invariant is evaluated before the loop;
- Variable hoisting, scope rule in JavaScript, where variables have function scope, and behave as if they were declared (but not defined) at the top of a function.

See also

- Basic block
- Block scope
- Closure (computer programming)
- Control flow

References

1. Perlis, A. J.; Samelson, K. (1958). "Preliminary report: international algebraic language". *Communications of the ACM*. New York, NY, USA: ACM. **1** (12): 8–22. doi:10.1145/377924.594925 (https://doi.org/10.1145%2F377924.594925).
2. Backus, J. W.; Bauer, F. L.; Green, J.; Katz, C.; McCarthy, J.; Perlis, A. J.; Rutishauser, H.; Samelson, K.; Vauquois, B.; Wegstein, J. H.; van Wijngaarden, A.; Woodger, M. (May 1960). Naur, Peter, ed. "Report on the Algorithmic Language ALGOL 60" (http://www.masswerk.at/algol60/report.htm). **3** (5). New York, NY, USA: ACM: 299–314. doi:10.1145/367236.367262 (https://doi.org/10.1145%2F367236.367262). ISSN 0001-0782 (https://www.worldcat.org/issn/0001-0782). Retrieved 2009-10-27.