Structured programming

Structured programming is a programming paradigm aimed at improving the clarity, quality, and development time of a computer program by making extensive use of subroutines, block structures, for and while loops—in contrast to using simple tests and jumps such as the *go to* statement, which could lead to "spaghetti code" that is difficult to follow and maintain.

It emerged in the late 1950s with the appearance of the ALGOL 58 and ALGOL 60 programming languages,[1] with the latter including support for block structures. Contributing factors to its popularity and widespread acceptance, at first in academia and later among practitioners, include the discovery of what is now known as the structured program theorem in 1966,[2] and the publication of the influential "Go To Statement Considered Harmful" open letter in 1968 by Dutch computer scientist Edsger W. Dijkstra, who coined the term "structured programming".[3]

Structured programming is most frequently used with deviations that allow for clearer programs in some particular cases, such as when exception handling has to be performed.

Contents

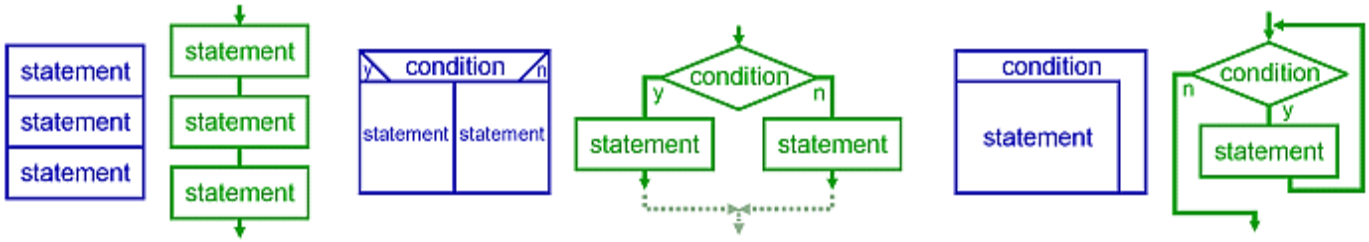
Elements

Control structures

Following the structured program theorem, all programs are seen as composed of control structures:

- "Sequence"; ordered statements or subroutines executed in sequence.
- "Selection"; one or a number of statements is executed depending on the state of the program. This is usually expressed with keywords such as `if..then..else..endif`.

- "Iteration"; a statement or block is executed until the program reaches a certain state, or operations have been applied to every element of a collection. This is usually expressed with keywords such as `while`, `repeat`, `for` or `do..until`. Often it is recommended that each loop should only have one entry point (and in the original structural programming, also only one exit point, and a few languages enforce this).
- "Recursion"; a statement is executed by repeatedly calling itself until termination conditions are met. While similar in practice to iterative loops, recursive loops may be more computationally efficient, and are implemented differently as a cascading stack.



Graphical representations of the three basic patterns using NS diagrams (blue) and flow charts (green).

Subroutines

Subroutines; callable units such as procedures, functions, methods, or subprograms are used to allow a sequence to be referred to by a single statement.

Blocks

Blocks are used to enable groups of statements to be treated as if they were one statement. *Block-structured* languages have a syntax for enclosing structures in some formal way, such as an if-statement bracketed by `if..fi` as in ALGOL 68, or a code section bracketed by `BEGIN..END`, as in PL/I, whitespace indentation as in Python - or the curly braces `{...}` of C and many later languages.

Structured programming languages

It is possible to do structured programming in any programming language, though it is preferable to use something like a procedural programming language. Some of the languages initially used for structured programming include: ALGOL, Pascal, PL/I and Ada, but most new procedural programming languages since that time have included features to encourage structured programming, and sometimes deliberately left out features – notably GOTO – in an effort to make unstructured programming more difficult. *Structured programming* (sometimes known as modular programming) enforces a logical structure on the program being written to make it more efficient and easier to understand and modify.

History

Theoretical foundation

The structured program theorem provides the theoretical basis of structured programming. It states that three ways of combining programs—sequencing, selection, and iteration—are sufficient to express any computable function. This observation did not originate with the structured programming movement; these structures are sufficient to describe the instruction cycle of a central processing unit, as well as the operation of a Turing machine. Therefore, a processor is always executing a "structured program" in this sense, even if the instructions it reads from memory are not part of a structured program. However, authors usually credit the result to a 1966 paper by Böhm and Jacopini, possibly because Dijkstra cited this paper himself.[4] The structured program theorem does not address how to write and analyze a usefully structured program. These issues were addressed during the late 1960s and early 1970s, with major contributions by Dijkstra, Robert W. Floyd, Tony Hoare, Ole-Johan Dahl, and David Gries.

Debate

P. J. Plauger, an early adopter of structured programming, described his reaction to the structured program theorem:

> Us converts waved this interesting bit of news under the noses of the unreconstructed assembly-language programmers who kept trotting forth twisty bits of logic and saying, 'I betcha can't structure this.' Neither the proof by Böhm and Jacopini nor our repeated successes at writing structured code brought them around one day sooner than they were ready to convince themselves.[5]

Donald Knuth accepted the principle that programs must be written with provability in mind, but he disagreed (and still disagrees) with abolishing the GOTO statement. In his 1974 paper, "Structured Programming with Goto Statements",[6] he gave examples where he believed that a direct jump leads to clearer and more efficient code without sacrificing provability. Knuth proposed a looser structural constraint: It should be possible to draw a program's flow chart with all forward branches on the left, all backward branches on the right, and no branches crossing each other. Many of those knowledgeable in compilers and graph theory have advocated allowing only reducible flow graphs.

Structured programming theorists gained a major ally in the 1970s after IBM researcher Harlan Mills applied his interpretation of structured programming theory to the development of an indexing system for the *New York Times* research file. The project was a great engineering success, and managers at other companies cited it in support of adopting structured programming, although Dijkstra criticized the ways that Mills's interpretation differed from the published work.

As late as 1987 it was still possible to raise the question of structured programming in a computer science journal. Frank Rubin did so in that year with an open letter titled ""GOTO considered harmful" considered harmful".[7] Numerous objections followed, including a response from Dijkstra that sharply criticized both Rubin and the concessions other writers made when responding to him.

Outcome

By the end of the 20th century nearly all computer scientists were convinced that it is useful to learn and apply the concepts of structured programming. High-level programming languages that originally lacked programming structures, such as FORTRAN, COBOL, and BASIC, now have them.

Common deviations

While goto has now largely been replaced by the structured constructs of selection (if/then/else) and repetition (while and for), few languages are purely structured. The most common deviation, found in many languages, is the use of a return statement for early exit from a subroutine. This results in multiple exit points, instead of the single exit point required by structured programming. There are other constructions to handle cases that are awkward in purely structured programming.

Early exit

The most common deviation from structured programming is early exit from a function or loop. At the level of functions, this is a `return` statement. At the level of loops, this is a `break` statement (terminate the loop) or `continue` statement (terminate the current iteration, proceed with next iteration). In structured programming, these can be replicated by adding additional branches or tests, but for returns from nested code this can add significant complexity. C is an early and prominent example of these constructs. Some newer languages also have "labeled breaks", which allow breaking out of more than just the innermost loop. Exceptions also allow early exit, but have further consequences, and thus are treated below.

Multiple exits can arise for a variety of reasons, most often either that the subroutine has no more work to do (if returning a value, it has completed the calculation), or has encountered "exceptional" circumstances that prevent it from continuing, hence needing exception handling.

The most common problem in early exit is that cleanup or final statements are not executed – for example, allocated memory is not deallocated, or open files are not closed, causing memory leaks or resource leaks. These must be done at each return site, which is brittle and can easily result in bugs. For instance, in later development, a return statement could be overlooked by a developer, and an

action which should be performed at the end of a subroutine (e.g., a trace statement) might not be performed in all cases. Languages without a return statement, such as standard Pascal don't have this problem.

Most modern languages provide language-level support to prevent such leaks;[8] see detailed discussion at resource management. Most commonly this is done via unwind protection, which ensures that certain code is guaranteed to be run when execution exits a block; this is a structured alternative to having a cleanup block and a `goto`. This is most often known as `try...finally`, and considered a part of exception handling. Various techniques exist to encapsulate resource management. An alternative approach, found primarily in C++, is Resource Acquisition Is Initialization, which uses normal stack unwinding (variable deallocation) at function exit to call destructors on local variables to deallocate resources.

Kent Beck, Martin Fowler and co-authors have argued in their refactoring books that nested conditionals may be harder to understand than a certain type of flatter structure using multiple exits predicated by guard clauses. Their 2009 book flatly states that "one exit point is really not a useful rule. Clarity is the key principle: If the method is clearer with one exit point, use one exit point; otherwise don't". They offer a cookbook solution for transforming a function consisting only of nested conditionals into a sequence of guarded return (or throw) statements, followed by a single unguarded block, which is intended to contain the code for the common case, while the guarded statements are supposed to deal with the less common ones (or with errors).[9] Herb Sutter and Andrei Alexandrescu also argue in their 2004 C++ tips book that the single-exit point is an obsolete requirement.[10]

In his 2004 textbook, David Watt writes that "single-entry multi-exit control flows are often desirable". Using Tennent's framework notion of sequencer, Watt uniformly describes the control flow constructs found in contemporary programming languages and attempts to explain why certain types of sequencers are preferable to others in the context of multi-exit control flows. Watt writes that unrestricted gotos (jump sequencers) are bad because the destination of the jump is not self-explanatory to the reader of a program until the reader finds and examines the actual label or address that is the target of the jump. In contrast, Watt argues that the conceptual intent of a return sequencer is clear from its own context, without having to examine its destination. Watt writes that a class of sequencers known as *escape sequencers*, defined as a "sequencer that terminates execution of a textually enclosing command or procedure", encompasses both breaks from loops (including multi-level breaks) and return statements. Watt also notes that while jump sequencers (gotos) have been somewhat restricted in languages like C, where the target must be an inside the local block or an encompassing outer block, that restriction alone is not sufficient to make the intent of gotos in C self-describing and so they can still produce "spaghetti code". Watt also examines how exception sequencers differ from escape and jump sequencers; this is explained in the next section of this article.[11]

In contrast to the above, Bertrand Meyer wrote in his 2009 textbook that instructions like `break` and `continue` "are just the old `goto` in sheep's clothing" and strongly advised against their use.[12]

Exception handling

Based on the coding error from the Ariane 501 disaster, software developer Jim Bonang argues that any exceptions thrown from a function violate the single-exit paradigm, and proposes that all inter-procedural exceptions should be forbidden. In C++ syntax, this is done by declaring all function signatures as `noexcept` (since C++11) or `throw()`.[13] Bonang proposes that all single-exit conforming C++ should be written along the lines of:

```
bool myCheck1() throw()
{
  bool success = false;

  try
  {
    // do something that may throw exceptions
    if(myCheck2() == false)
    {
        throw SomeInternalException();
    }

    // other code similar to the above
    success = true;
  }
```

```
  catch(...)
  {
      // all exceptions caught and logged
  }

  return success;
}
```

Peter Ritchie also notes that, in principle, even a single `throw` right before the `return` in a function constitutes a violation of the single-exit principle, but argues that Dijkstra's rules were written in a time before exception handling became a paradigm in programming languages, so he proposes to allow any number of throw points in addition to a single return point. He notes that solutions which wrap exceptions for the sake of creating a single-exit have higher nesting depth and thus are more difficult to comprehend, and even accuses those who propose to apply such solutions to programming languages which support exceptions of engaging in cargo cult thinking.[14]

David Watt also analyzes exception handling in the framework of sequencers (introduced in this article in the previous section on early exits.) 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!" He 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 code to willfully ignore it. Based on these arguments, Watt concludes that jump sequencers or escape sequencers (discussed in the previous section) aren't as suitable as a dedicated exception sequencer with the semantics discussed above.[15]

The textbook by Louden and Lambert emphasizes that exception handling differs from structured programming constructs like `while` loops because the transfer of control "is set up at a different point in the program than that where the actual transfer takes place. At the point where the transfer actually occurs, there may be no syntactic indication that control will in fact be transferred."[16] Computer science professor Arvind Kumar Bansal also notes that in languages which implement exception handling, even control structures like `for`, which have the single-exit property in absence of exceptions, no longer have it in presence of exceptions, because an exception can prematurely cause an early exit in any part of the control structure; for instance if `init()` throws an exception in `for (init(); check(); increm())`, then the usual exit point after check() is not reached.[17] Citing multiple prior studies by others (1999-2004) and their own results, Westley Weimer and George Necula wrote that a significant problem with exceptions is that they "create hidden control-flow paths that are difficult for programmers to reason about".[18]:8:27

The necessity to limit code to single-exit points appears in some contemporary programming environments focused on parallel computing, such as OpenMP. The various parallel constructs from OpenMP, like `parallel do`, do not allow early exits from inside to the outside of the parallel construct; this restriction includes all manner of exits, from `break` to C++ exceptions, but all of these are permitted inside the parallel construct if the jump target is also inside it.[19]

Multiple entry

More rarely, subprograms allow multiple *entry*. This is most commonly only *re*-entry into a coroutine (or generator/semicoroutine), where a subprogram yields control (and possibly a value), but can then be resumed where it left off. There are a number of common uses of such programming, notably for streams (particularly input/output), state machines, and concurrency. From a code execution point of view, yielding from a coroutine is closer to structured programming than returning from a subroutine, as the subprogram has not actually terminated, and will continue when called again – it is not an early exit. However, coroutines mean that multiple subprograms have execution state – rather than a single call stack of subroutines – and thus introduce a different form of complexity.

It is very rare for subprograms to allow entry to an arbitrary position in the subprogram, as in this case the program state (such as variable values) is uninitialized or ambiguous, and this is very similar to a goto.

State machines

Some programs, particularly parsers and communications protocols, have a number of states that follow each other in a way that is not easily reduced to the basic structures, and some programmers implement the state-changes with a jump to the new state. This type of state-switching is often used in the Linux kernel.

However, it is possible to structure these systems by making each state-change a separate subprogram and using a variable to indicate the active state (see trampoline). Alternatively, these can be implemented via coroutines, which dispense with the trampoline.

See also

- DRAKON
- Minimal evaluation
- Nassi–Shneiderman diagram
- Structure chart
- Switch statement

Notes

1. Clark, Leslie B. Wilson, Robert G.; Robert, Clark (2000). *Comparative programming languages* (https://books.google.co.uk/books?id=bVSjoO8f2fMC&lpg=PR11&ots=zItCa2GxG6&dq=Comparative%20Programming%20Languages&lr&pg=PA20#v=onepage&q=Comparative%20Programming%20Languages&f=false) (3rd ed.). Harlow, England: Addison-Wesley. p. 20. ISBN 9780201710120. Retrieved 25 November 2015.
2. Bohm, Corrado; Giuseppe Jacopini (May 1966). "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules" (http://www.cs.unibo.it/~martini/PP/bohm-jac.pdf) (PDF). *Communications of the ACM*. **9** (5): 366–371. doi:10.1145/355592.365646 (https://doi.org/10.1145%2F355592.365646).
3. Dijkstra 1968, "The unbridled use of the go to statement has as an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. ... The go to statement as it stands is just too primitive, it is too much an invitation to make a mess of one's program."
4. Dijkstra, E. W. (March 1968). "Letters to the editor: go to statement considered harmful". *Communications of the ACM*. **11** (3): 147–148. ISSN 0001-0782 (https://www.worldcat.org/issn/0001-0782). doi:10.1145/362929.362947 (https://doi.org/10.1145%2F362929.362947).
5. Plauger, P. J. (February 12, 1993). *Programming on Purpose, Essays on Software Design* (1 ed.). Prentice-Hall. p. 25. ISBN 978-0-13-721374-0.
6. Donald Knuth - Structured programming with go to statements (http://cs.sjsu.edu/~mak/CS185C/KnuthStructuredProgrammingGoTo.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. Elder, Jackson & Liblit 2008.
9. Jay Fields; Shane Harvie; Martin Fowler; Kent Beck (2009). *Refactoring: Ruby Edition*. Pearson Education. pp. 274–279. ISBN 978-0-321-60350-0.
10. Herb Sutter; Andrei Alexandrescu (2004). *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Pearson Education. ISBN 978-0-13-265442-5. ""Example 4: Single entry, single exit ("SESE"). Historically, some coding standards have required that each function have exactly one exit, meaning one return statement. Such a requirement is obsolete in languages that support exceptions and destructors, where functions typically have numerous implicit exits.")"
11. David Anthony Watt; William Findlay (2004). *Programming language design concepts*. John Wiley & Sons. pp. 215–221. ISBN 978-0-470-85320-7.
12. 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.
13. http://pragprog.com/magazines/2012-04/the-pragmatic-defense

14. http://msmvps.com/blogs/peterritchie/archive/2008/03/07/single-entry-single-exit-should-it-still-be-applicable-in-object-oriented-languages.aspx
15. David Anthony Watt; William Findlay (2004). *Programming language design concepts*. John Wiley & Sons. pp. 221–222. ISBN 978-0-470-85320-7.
16. Kenneth C. Louden; Kenneth A. Lambert (2011). *Programming Languages: Principles and Practices* (3 ed.). Cengage Learning. p. 423. ISBN 1-111-52941-8.
17. Arvind Kumar Bansal (2013). *Introduction to Programming Languages*. CRC Press. p. 135. ISBN 978-1-4665-6514-2.
18. Weimer, W & Necula, G.C. (2008). "Exceptional Situations and Program Reliability" (http://www.cs.virginia.edu/~weimer/p/weimer-toplas2008.pdf) (PDF). *ACM Transactions on Programming Languages and Systems, vol 30 (2)*.
19. Rohit Chandra (2001). *Parallel Programming in OpenMP*. Morgan Kaufmann. p. 45. ISBN 978-1-55860-671-5.

References

- Edsger Dijkstra, *Notes on Structured Programming* (http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF), pg. 6
- Böhm, C.; Jacopini, G. (May 1966). "Flow diagrams, Turing machines and languages with only two formation rules" (http://www.cs.unibo.it/~martini/PP/bohm-jac.pdf) (PDF). *Communications of the ACM*. **9** (5): 366–371. doi:10.1145/355592.365646 (https://doi.org/10.1145%2F355592.365646).
- 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).
- Michael A. Jackson, *Principles of Program Design*, Academic Press, London, 1975.
- O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare *Structured Programming*, Academic Press, London, 1972 ISBN 0-12-200550-3
  - this volume includes an expanded version of the *Notes on Structured Programming*, above, including an extended example of using the structured approach to develop a backtracking algorithm to solve the 8 Queens problem.
  - a pdf version is in the ACM Classic Books Series (http://portal.acm.org/citation.cfm?id=1243380&jmp=cit&coll=portal&dl=GUIDE&CFID=://www.acm.org/publications/&CFTOKEN=www.acm.org/publications/#CIT)
  - Note that the third chapter of this book, by Dahl, describes an approach which is easily recognized as Object Oriented Programming. It can be seen as another way to "usefully structure" a program to aid in showing that it is correct.
- Elder, Matt; Jackson, Steve; Liblit, Ben (October 2008). *Code Sandwiches* (http://research.cs.wisc.edu/techreports/2008/TR1647.pdf) (PDF) (Technical report). University of Wisconsin–Madison. 1647, abstract (http://pages.cs.wisc.edu/~liblit/tr-1647/)

External links

- BPStruct (http://code.google.com/p/bpstruct/) - A tool to structure concurrent systems (programs, process models)
- J. Darlinton; M. Ghanem; H. W. To (1993), "Structured Parallel Programming" (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.4610), *In Programming Models for Massively Parallel Computers. IEEE Computer Society Press. 1993*