WIKIPEDIA

Imperative programming

In computer science, **imperative programming** is a programming paradigm that uses statements that change a program's state. In much the same way that the imperative mood in natural languages expresses commands, an imperative program consists of commands for the computer to perform. Imperative programming focuses on describing *how* a program operates.

The term is often used in contrast to declarative programming, which focuses on *what* the program should accomplish without specifying *how* the program should achieve the result.

Contents

Imperative and procedural programming

Procedural programming is a type of imperative programming in which the program is built from one or more procedures (also termed subroutines or functions). The terms are often used as synonyms, but the use of procedures has a dramatic effect on how imperative programs appear and how they are constructed. Heavily-procedural programming, in which state changes are localized to procedures or restricted to explicit arguments and returns from procedures, is a form of structured programming. From the 1960s onwards, structured programming and modular programming in general have been promoted as techniques to improve the maintainability and overall quality of imperative programs. The concepts behind object-oriented programming attempt to extend this approach.[1]

Procedural programming could be considered a step towards declarative programming. A programmer can often tell, simply by looking at the names, arguments, and return types of procedures (and related comments), what a particular procedure is supposed to do, without necessarily looking at the details of how it achieves its result. At the same time, a complete program is still imperative since it *fixes* the statements to be executed and their order of execution to a large extent.

Rationale and foundations of imperative programming

The hardware implementation of almost all computers is imperative.[note 1] Nearly all computer hardware is designed to execute machine code, which is native to the computer and is written in the imperative style. From this low-level perspective, the program state is defined by the contents of memory, and the statements are instructions in the native machine language of the computer. Higher-level imperative languages use variables and more complex statements, but still follow the same paradigm. Recipes and process checklists, while not computer programs, are also familiar concepts that are similar in style to imperative programming; each step is an instruction, and the physical world holds the state. Since the basic ideas of imperative programming are both conceptually familiar and directly embodied in the hardware, most computer languages are in the imperative style.

Assignment statements, in imperative paradigm, perform an operation on information located in memory and store the results in memory for later use. High-level imperative languages, in addition, permit the evaluation of complex expressions, which may consist of a combination of arithmetic operations and function evaluations, and the assignment of the resulting value to memory. Looping statements (as in while loops, do while loops, and for loops) allow a sequence of statements to be executed multiple times. Loops can either execute the statements they contain a predefined number of times, or they can execute them repeatedly until some condition changes. Conditional branching statements allow a sequence of statements to be executed only if some condition is met. Otherwise, the statements are skipped and the execution sequence continues from the statement following them. Unconditional branching statements allow an execution sequence to be transferred to another part of a program. These include the jump (called *goto* in many languages), switch, and the subprogram, subroutine, or procedure call (which usually returns to the next statement after the call).

Early in the development of high-level programming languages, the introduction of the block enabled the construction of programs in which a group of statements and declarations could be treated as if they were one statement. This, alongside the introduction of subroutines, enabled complex structures to be expressed by hierarchical decomposition into simpler procedural structures.

Many imperative programming languages (such as Fortran, BASIC, and C) are abstractions of assembly language.[2]

History of imperative and object-oriented languages

The earliest imperative languages were the machine languages of the original computers. In these languages, instructions were very simple, which made hardware implementation easier, but hindered the creation of complex programs. FORTRAN, developed by John Backus at International Business Machines (IBM) starting in 1954, was the first major programming language to remove the obstacles presented by machine code in the creation of complex programs. FORTRAN was a compiled language that allowed named variables, complex expressions, subprograms, and many other features now common in imperative languages. The next two decades saw the development of many other major high-level imperative programming languages. In the late 1950s and 1960s, ALGOL was developed in order to allow mathematical algorithms to be more easily expressed, and even served as the operating system's target language for some computers. MUMPS (1966) carried the imperative paradigm to a logical extreme, by not having any statements at all, relying purely on commands, even to the extent of making the IF and ELSE commands independent of each other, connected only by an intrinsic variable named $TEST. COBOL (1960) and BASIC (1964) were both attempts to make programming syntax look more like English. In the 1970s, Pascal was developed by Niklaus Wirth, and C was created by Dennis Ritchie while he was working at Bell Laboratories. Wirth went on to design Modula-2 and Oberon. For the needs of the United States Department of Defense, Jean Ichbiah and a team at Honeywell began designing Ada in 1978, after a 4-year project to define the requirements for the language. The specification was first published in 1983, with revisions in 1995, 2005 and 2012.

The 1980s saw a rapid growth in interest in object-oriented programming. These languages were imperative in style, but added features to support objects. The last two decades of the 20th century saw the development of many such languages. Smalltalk-80, originally conceived by Alan Kay in 1969, was released in 1980, by the Xerox Palo Alto Research Center (PARC). Drawing from concepts in another object-oriented language—Simula (which is considered the world's first object-oriented programming language, developed in the 1960s)—Bjarne Stroustrup designed C++, an object-oriented language based on C. Design of C++ began in 1979 and the first implementation was completed in 1983. In the late 1980s and 1990s, the notable imperative languages drawing on object-oriented concepts were Perl, released by Larry Wall in 1987;

Python, released by Guido van Rossum in 1990; Visual Basic and Visual C++ (which included Microsoft Foundation Class Library (MFC) 2.0), released by Microsoft in 1991 and 1993 respectively; PHP, released by Rasmus Lerdorf in 1994; Java, released by Sun Microsystems in 1994, JavaScript, by Brendan Eich (Netscape), and Ruby, by Yukihiro "Matz" Matsumoto, both released in 1995. Microsoft's .NET Framework (2002) is imperative at its core, as are its main target languages, VB.NET and C# that run on it; however Microsoft's F#, a functional language, also runs on it.

See also

- Functional programming
- Comparison of programming paradigms
- Reactive programming
- History of programming languages
- List of imperative programming languages

Notes

1. Reconfigurable computing is a notable exception.

References

1. Michael Stevens (2011). "Programming paradigms and an overview of C - COMP3610 - Principles of Programming Languages: Object-Oriented programming" (http://cs.anu.edu.au/student/comp3610/lectures/Paradigms.pdf) (PDF). Australian National University. p. 5. Archived from the original (http://cs.anu.edu.au/) on 2011. Retrieved 2012-05-22. "Object oriented programming extends the concept of modularity by introducing objects"
2. Bruce Eckel (2006). *Thinking in Java* (https://books.google.com/books?id=bQVvAQAAQBAJ&pg=PA24&lpg=PA24&dq=imperative&source=bl&ots=LXUh8GZ17E&sig=u2aOh64yjWEYX_nCs9NolKmKhE4&hl=en&sa=X&ei=jLv0U6i1IMKSyAS-7oKADg&ved=0CDUQ6AEwAw#v=onepage&q=imperative&f=false). Pearson Education. p. 24. ISBN 978-0-13-187248-6.

- Pratt, Terrence W. and Marvin V. Zelkowitz. *Programming Languages: Design and Implementation*, 3rd ed. Englewood Cliffs, N.J.: Prentice Hall, 1996.
- Sebesta, Robert W. *Concepts of Programming Languages*, 3rd ed. Reading, Mass.: Addison-Wesley Publishing Company, 1996.

Originally based on the article 'Imperative programming' by Stan Seibert, from Nupedia, licensed under the GNU Free Documentation License.