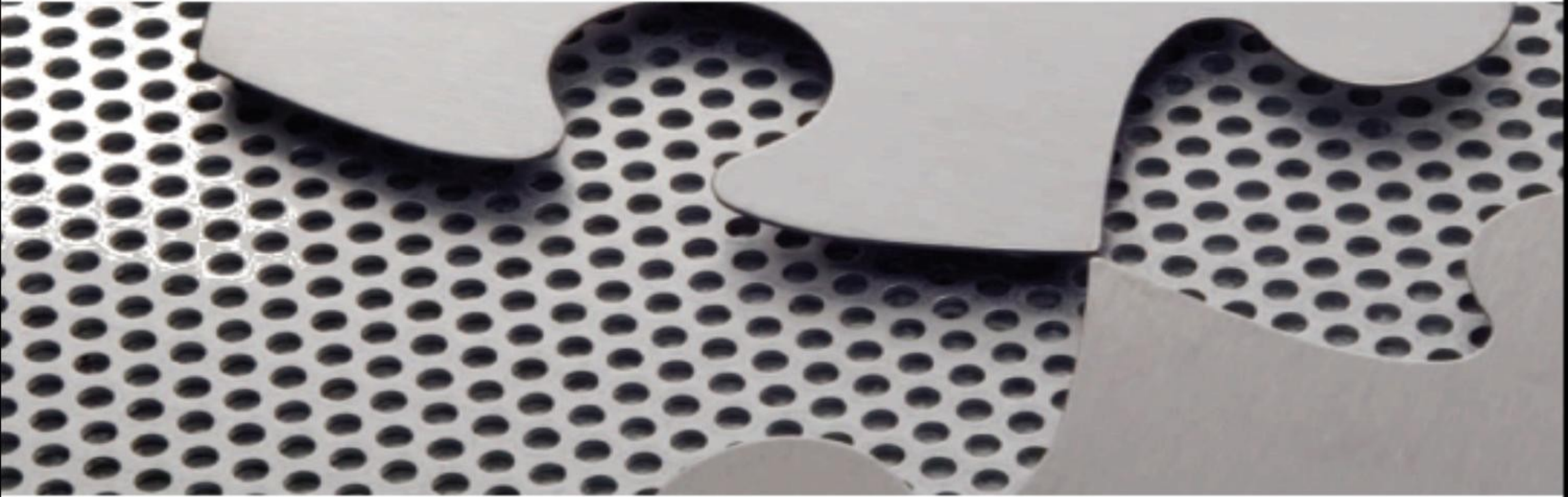
Programming Languages
Third Edition



Chapter 2

Language Design Criteria

Objectives

- Describe the history of programming language design criteria
- Understand efficiency in programming languages
- Understand regularity in programming languages
- Understand security in programming languages
- Understand extensibility in programming languages
- Understand the design goals of C++
- Understand the design goals of Python

Background

- What is good programming language design?
- What criteria should be used to judge a language?
- How should success or failure of a language be defined?
- We will define a language as successful if it satisfies any or all of these criteria:
  - It achieves the goals of its designers
  - It attains widespread use in an application area
  - It serves as a model for other languages that are successful

Background (cont'd.)

- When creating a new language, decide on an overall goal and keep it in mind throughout the design process
- This is especially important for special purpose languages
  - The abstractions for the target application area must be built into the language design
- This chapter introduces some general design criteria and presents a set of detailed principles as potential aids to the designer

Historical Overview

- In the early days, machines were extremely slow and memory was scarce
  - Program speed and memory usage were prime concerns
- **Efficiency of execution**: primary design criterion
  - Early FORTRAN code more or less directly mapped to machine code, minimizing the amount of translation required by the compiler
- **Writability**: the quality of a language that enables a programmer to use it to express computation clearly, correctly, concisely, and quickly

Historical Overview (cont'd.)

- In the early days, writability was less important than efficiency
- Algol60 was designed for expressing algorithms in a logically clear and concise way
  - Incorporated block structure, structured control statements, a more structured array type, and recursion
- COBOL attempted to improve readability of programs by trying to make them look like ordinary English
  - However, this made them long and verbose

Historical Overview (cont'd.)

- In the 1970s and early 1980s, the emphasis was on simplicity and abstraction, along with reliability
  - Mathematical definitions for language constructs were introduced, along with mechanisms to allow a translator to partially prove the correctness of a program before translation
  - This led to strong data typing
- In the 1980s and 1990s, the emphasis was on logical or mathematical precision
  - This led to a renewed interest in functional languages

Historical Overview (cont'd.)

- The most influential design criteria of the last 25 years is the object-oriented approach to abstraction
  - Led to the use of libraries and other object-oriented techniques to increase reusability of existing code
- In addition to the early goals of efficiency, nearly every design decision still considers readability, abstraction, and complexity control

Efficiency

- **Efficiency**: usually thought of as efficiency of the target code
- Example: strong data typing, enforced at compile time, means that the runtime does not need to check the data types before executing operations
- Example: early FORTRAN required that all data declarations and subroutine calls had to be known at compile time to allow the memory space to be allocated once at beginning of execution

Efficiency (cont'd.)

- **Programmer efficiency**: how quickly and easily can a person read and write in the programming language?
- **Expressiveness**: how easy is it to express complex processes and structures?
- Conciseness of the syntax also contributes to programmer efficiency
  - Example: Python does not require braces or semi-colons, only indentation and the colon (:)

Efficiency (cont'd.)

- Reliability of a program can be viewed as an efficiency issue
  - Unreliable programs require programmer time to diagnose and correct
- Programmer efficiency is also impacted by the ease with which errors can be found and corrected
- Since roughly 90% of time is spent on debugging and maintaining programs, maintainability may be the most important index of programming language efficiency

Regularity

- **Regularity**: refers to how well the features of a language are integrated
- Greater regularity implies:
  - Fewer restrictions on the use of particular constructs
  - Fewer strange interactions between constructs
  - Fewer surprises in general in the way the language features behave
- Languages that satisfy the criterion of regularity are said to adhere to the principle of least astonishment

Regularity (cont'd)

- Regularity can be subdivided into three concepts:
  - Generality
  - Orthogonal design
  - Uniformity
- **Generality**: achieved by avoiding special cases in the availability or use of constructs and by combining closely related constructs into a single more general one
- **Orthogonal design**: constructs can be combined in any meaningful way, with no unexpected restrictions or behaviors

Regularity (cont'd)

- **Uniformity**: a design in which similar things look similar and have similar meanings while different things look different
- Can classify a feature or construct as irregular if it lacks one of these three qualities

Generality

- **Generality**: A language with this property avoids special cases wherever possible
- Example: procedures and functions
  - Pascal allows nesting of functions and procedures and passing of functions and procedures as parameters to other functions and procedures but does not allow them to be assigned to variables or stored in data structures
- Example: operators
  - In C, cannot directly compare two structures with ==; thus, this operator lacks generality

Generality (cont'd.)

- Example: constants
  - Pascal does not allow the value assigned to constants to be computed by expressions, while Ada has a completely general constant declaration facility

Orthogonality

- In a language that is truly orthogonal, constructs do not behave differently in different contexts
  - Restrictions that are context dependent are nonorthogonal, while restrictions that apply regardless of context exhibit a lack of generality
- Example: function return types
  - Pascal allows only scalar or pointer types as return values
  - C and C++ allow values of all data types except array types
  - Ada and Python allow all data types

Orthogonality (cont'd.)

- Example: placement of variable declarations
  - C requires that local variables be defined only at the beginning of a block
  - C++ allows variable definitions at any point inside a block prior to use
- Example: **primitive** and **reference** types
  - In Java, primitive types use **value semantics** (values are copied during assignment), while object types (or reference types) use **reference semantics** (assignment produces two references to the same object)

Orthogonality (cont’d.)

- Orthogonality was a major design goal of Algol68
  - It is still the best example of a language in which constructs can be combined in all meaningful ways

Uniformity

- **Uniformity**: refers to the consistency of appearance and behavior of language constructs
- Example: extra semicolon
  - C++ requires a semicolon after a class definition but forbids its use after a function definition
- Example: using assignment to return a value
  - Pascal uses the function name in an assignment statement to return the function's value
    - Looks confusingly like a standard assignment statement
  - Other languages use a return statement

Causes of Irregularities

- Many irregularities are case studies in the difficulties of language design
- Example: extra semicolon problem in C++ was a byproduct of the need to be compatible with C
- Example: irregularity of primitive types and reference types in Java is the result of the designer’s concern with efficiency
- It is possible to focus too much on a particular goal
- Example: Algol68 met its goals of generality and orthogonality, but this led to a somewhat obscure and complex language

Security

- Reliability can be affected if restrictions are not imposed on certain features
  - Pascal: pointers are restricted to reduce security problems
  - C: pointers are much less restricted and thus more prone to misuse and error
  - Java: pointers were eliminated altogether (they are implicit in object allocation), but Java requires a more complicated runtime environment
- **Security**: closely related to reliability

Security (cont'd.)

- A language designed with security in mind:
  - Discourages programming errors
  - Allows errors to be discovered and reported
- Types, type-checking, and variable declarations resulted from a concern for security
- Exclusive focus on security can compromise the expressiveness and conciseness of a language
  - Typically forces the programmer to laboriously specify as many things as possible in the code

Security (cont'd.)

- ML and Haskell are functional languages that attempt to be secure yet allow for maximum expressiveness and generality
  - They allow multityped objects, do not require declarations, and yet perform static type-checking
- **Semantically safe**: languages that prevent a programmer from compiling or executing any statements or expressions that violate the language definition
  - Examples: Python, Lisp, Java

Extensibility

- **Extensible language**: a language that allows the user to add features to it
- Example: the ability to define new data types and new operations (functions or procedures)
- Example: new releases that extend the built-in features of the language
- Very few languages allow additions to the syntax and semantics
  - Lisp allows new syntax and semantics via a macro
- **Macro**: specifies the syntax of a piece of code that expands to other standard code when compiled

C++: An Object-Oriented Extension of C

- C++: created by Bjarne Stroustrup at Bell Labs in 1979-80
- He chose to base his new language on C because of its:
  - Flexibility
  - Efficiency
  - Availability
  - Portability
- He chose to add the class construct from Simula67 language

C++: An Object-Oriented Extension of C (cont'd.)

- Design goals for C++:
  - Support for good program development in the form of classes, inheritance, and strong type-checking
  - Efficient execution on the order of C or BCPL
  - Highly portable, easily implemented, and easily interfaced with other tools

C++: First Implementations

- First implementation in 1979-80 in the form of a preprocessor called Cpre, which generated ordinary C code
- 1985: replaced the preprocessor with a more sophisticated compiler (which still generated C code for portability)
  - Compiler was called **Cfront**
  - Language was now called C++
  - Added dynamic binding of methods, type parameters, and general overloading

C++: First Implementations (cont'd.)

- Design goals for C++:
  - Maintain C compatibility as far as practical
  - Should undergo incremental development based firmly in practical experience
  - Any added feature must not degrade runtime efficiency or affect existing programs negatively
  - Should not force any one style of programming
  - Should maintain and strengthen its type-checking
  - Should be learnable in stages
  - Should maintain compatibility with other systems and languages

C++: Growth

- Cpre and Cfront were distributed for educational purposes at no cost, creating interest in the language
- 1986: first commercial implementation
- Success of the language indicated that a concerted effort at creating a standard language was necessary

C++: Standardization

- Because C++ was rapidly growing in use, was continuing to evolve, and had several different implementations, standardization was a problem
- 1989: Stroustrup produced a reference manual
- 1990-1991: ANSI and ISO standards committees accepted the manual as the base document for the standardization effort
- 1994: addition of a standard library of containers and algorithms
- 1998: proposed standards became the actual ANSI/ISO standard

C++: Retrospective

- Why was C++ a success?
  - Introduced just as interest in object-oriented techniques was exploding
  - Straightforward syntax, not tied to any operating environment
  - Its semantics incurred no performance penalty
  - Its flexibility, hybrid nature, and its designer's willingness to extend its features were popular
- Detractors consider C++ to have too many features and too many ways of doing similar things

Python: A General-Purpose Scripting Language

- Guido van Rossum developed a translator and virtual machine for a scripting language called Python in 1986
- One of his goals was to allow Python to act as a bridge between system languages such as C and shell or scripting languages such as Perl
- Included a dictionary, a set of key/value pairs implemented via hashing, that was useful for representing collections of objects organized by content or association instead of by position

Python: Simplicity, Regularity, and Extensibility

- Design goals included:
  - A simple regular syntax
  - A set of powerful data types and libraries
  - Easy to use by novices

Python: Interactivity and Portability

- Python was designed for users who do not typically write large systems but instead write short programs
  - Development cycle provides immediate feedback with minimal overhead for I/O operations
- Python can be run in two modes:
  - Expressions or statements can be run in a Python shell for maximum interactivity
  - Can be composed into longer scripts saved in files and run from a terminal command prompt

Python: Interactivity and Portability (cont'd.)

- Design goal of portability was accomplished in two ways:
  - Python compiler translates source code to machine-independent byte code, which is run on a Python virtual machine (PVM)
  - Application-specific libraries support programs that must access databases, networks, the Web, GUI, and other resources and technologies

Python: Dynamic Typing vs. Finger Typing

- Python incorporates the dynamic typing mechanism found in Lisp and Smalltalk
  - All variables are untyped
  - Any variable can name any thing, but all things or values have a type
  - Type-checking occurs at runtime
- This results in less overhead for the programmer
  - Less “finger typing”
  - Programmer can get a code segment up and running much faster

Python: Retrospective

- Python was not intended to replace C or C++ for large or time-critical systems
- Runtime type-checking is not suitable for time-critical applications
- The absence of static type-checking can be a liability in testing and verification of a large software system
- Its design goal of ease of use for a novice or nonprogrammer has largely been achieved