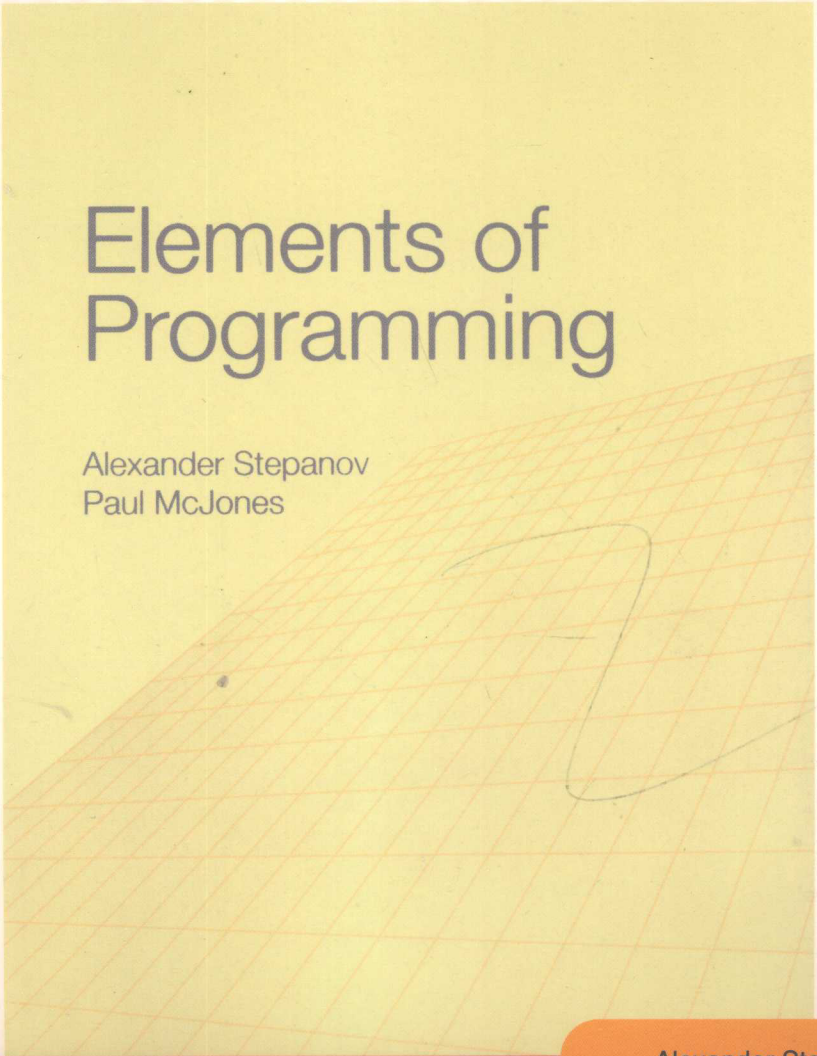
HZ BOOKS

经 典 原 版 书 库

PEARSON

编程的本质

（英文版）



（美）Alexander Stepanov
Paul McJones 著

机械工业出版社
China Machine Press

编程的本质

（英文版）

Elements of Programming

（美）Alexander Stepanov
Paul McJones 著

机械工业出版社
China Machine Press

本书版权登记号：图字：01-2010-1033

图书在版编目（CIP）数据

编程的本质（英文版）/（美）斯特潘诺夫（Stepanov, A.）等著. —北京：机械工业出版社，2010.3

（经典原版书库）

书名原文：Elements of Programming

ISBN 978-7-111-30027-4

Ⅰ. 编…　Ⅱ. 斯…　Ⅲ. 程序设计-英文　Ⅳ. TP311.1

中国版本图书馆CIP数据核字（2010）第038087号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）

责任编辑：李俊竹

北京瑞德印刷有限公司印刷

2010年3月第1版第1次印刷

170mm × 242mm · 17.25印张

标准书号：ISBN 978-7-111-30027-4

定价：49.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991，88361066

购书热线：（010）68326294，88379649，68995259

投稿热线：（010）88379604

读者信箱：hzjsj@hzbook.com

Preface

This book applies the deductive method to programming by affiliating programs with the abstract mathematical theories that enable them to work. Specification of these theories, algorithms written in terms of these theories, and theorems and lemmas describing their properties are presented together. The implementation of the algorithms in a real programming language is central to the book. While the specifications, which are addressed to human beings, should, and even must, combine rigor with appropriate informality, the code, which is addressed to the computer, must be absolutely precise even while being general.

As with other areas of science and engineering, the appropriate foundation of programming is the deductive method. It facilitates the decomposition of complex systems into components with mathematically specified behavior. That, in turn, is a necessary precondition for designing efficient, reliable, secure, and economical software.

The book is addressed to those who want a deeper understanding of programming, whether they are full-time software developers, or scientists and engineers for whom programming is an important part of their professional activity.

The book is intended to be read from beginning to end. Only by reading the code, proving the lemmas, and doing the exercises can readers gain understanding of the material. In addition, we suggest several projects, some open-ended. While the book is terse, a careful reader will eventually see the connections between its parts and the reasons for our choice of material. Discovering the architectural principles of the book should be the reader's goal.

We assume an ability to do elementary algebraic manipulations.[1] We also assume familiarity with the basic vocabulary of logic and set theory at the level of undergraduate courses on discrete mathematics; Appendix A summarizes the notation that we use. We provide definitions of a few concepts of abstract algebra when they are

1. For a refresher on elementary algebra, we recommend Chrystal [1904].

needed to specify algorithms. We assume programming maturity and understanding of computer architecture[2] and fundamental algorithms and data structures.[3]

We chose C++ because it combines powerful abstraction facilities with faithful representation of the underlying machine.[4] We use a small subset of the language and write requirements as structured comments. We hope that readers not already familiar with C++ are able to follow the book. Appendix B specifies the subset of the language used in the book.[5] Wherever there is a difference between mathematical notation and C++, the typesetting and the context determine whether the mathematical or C++ meaning applies. While many concepts and programs in the book have parallels in STL (the C++ Standard Template Library), the book departs from some of the STL design decisions. The book also ignores issues that a real library, such as STL, has to address: namespaces, visibility, inline directives, and so on.

Chapter 1 describes values, objects, types, procedures, and concepts. Chapters 2–5 describe algorithms on algebraic structures, such as semigroups and totally ordered sets. Chapters 6–11 describe algorithms on abstractions of memory. Chapter 12 describes objects containing other objects. The Afterword presents our reflections on the approach presented by the book.

Acknowledgments

We are grateful to Adobe Systems and its management for supporting the Foundations of Programming course and this book, which grew out of it. In particular, Greg Gilley initiated the course and suggested writing the book; Dave Story and then Bill Hensler provided unwavering support. Finally, the book would not have been possible without Sean Parent's enlightened management and continuous scrutiny of the code and the text. The ideas in the book stem from our close collaboration, spanning almost three decades, with Dave Musser. Bjarne Stroustrup deliberately evolved C++ to support these ideas. Both Dave and Bjarne were kind enough to come to San Jose and carefully review the preliminary draft. Sean Parent and Bjarne Stroustrup wrote the appendix defining the C++ subset used in the book. Jon Brandt reviewed multiple drafts of the book. John Wilkinson carefully read the final manuscript, providing innumerable valuable suggestions.

2. We recommend Patterson and Hennessy [2007].
3. For a selective but incisive introduction to algorithms and data structures, we recommend Tarjan [1983].
4. The standard reference is Stroustrup [2000].
5. The code in the book compiles and runs under Microsoft Visual C++ 9 and g++ 4. This code, together with a few trivial macros that enable it to compile, as well as unit tests, can be downloaded from www.elementsofprogramming.com.

The book has benefited significantly from the contributions of our editor, Peter Gordon, our project editor, Elizabeth Ryan, our copy editor, Evelyn Pyle, and the editorial reviewers: Matt Austern, Andrew Koenig, David Musser, Arch Robison, Jerry Schwarz, Jeremy Siek, and John Wilkinson.

We thank all the students who took the course at Adobe and an earlier course at SGI for their suggestions. We hope we succeeded in weaving the material from these courses into a coherent whole. We are grateful for comments from Dave Abrahams, Andrei Alexandrescu, Konstantine Arkoudas, John Banning, Hans Boehm, Angelo Borsotti, Jim Dehnert, John DeTreville, Boris Fomitchev, Kevlin Henney, Jussi Ketonen, Karl Malbrain, Mat Marcus, Larry Masinter, Dave Parent, Dmitry Polukhin, Jon Reid, Mark Ruzon, Geoff Scott, David Simons, Anna Stepanov, Tony Van Eerd, Walter Vannini, Tim Winkler, and Oleg Zabluda.

Finally, we are grateful to all the people who taught us through their writings or in person, and to the institutions that allowed us to deepen our understanding of programming.

About the Authors

Alexander Stepanov studied mathematics at Moscow State University from 1967 to 1972. He has been programming since 1972: first in the Soviet Union and, after emigrating in 1977, in the United States. He has programmed operating systems, programming tools, compilers, and libraries. His work on foundations of programming has been supported by GE, Brooklyn Polytechnic, AT&T, HP, SGI, and, since 2002, Adobe. In 1995 he received the *Dr. Dobb's Journal* Excellence in Programming Award for the design of the C++ Standard Template Library.

Paul McJones studied engineering mathematics at the University of California, Berkeley, from 1967 to 1971. He has been programming since 1967 in the areas of operating systems, programming environments, transaction processing systems, and enterprise and consumer applications. He has been employed by the University of California, IBM, Xerox, Tandem, DEC, and, since 2003, Adobe. In 1982 he and his coauthors received the ACM Programming Systems and Languages Paper Award for their paper "The Recovery Manager of the System R Database Manager."

Contents

Chapter 1
Foundations

Starting with a brief taxonomy of ideas, we introduce notions of value, object, type, procedure, *and* concept *that represent different categories of ideas in the computer. A central notion of the book,* regularity, *is introduced and elaborated. When applied to procedures, regularity means that procedures return equal results for equal arguments. When applied to types, regularity means that types possess the equality operator and equality-preserving copy construction and assignment. Regularity enables us to apply* equational reasoning *(substituting equals for equals) to transform and optimize programs.*

1.1 Categories of Ideas: Entity, Species, Genus

In order to explain what objects, types, and other foundational computer notions are, it is useful to give an overview of some categories of ideas that correspond to these notions.

An *abstract entity* is an individual thing that is eternal and unchangeable, while a *concrete entity* is an individual thing that comes into and out of existence in space and time. An *attribute*—a correspondence between a concrete entity and an abstract entity—describes some property, measurement, or quality of the concrete entity. *Identity*, a primitive notion of our perception of reality, determines the sameness of a thing changing over time. Attributes of a concrete entity can change without affecting its identity. A *snapshot* of a concrete entity is a complete collection of its attributes at a particular point in time. Concrete entities are not only physical entities but also legal, financial, or political entities. Blue and 13 are examples of abstract entities. Socrates and the United States of America are examples of concrete entities. The color of Socrates' eyes and the number of U.S. states are examples of attributes.

An *abstract species* describes common properties of essentially equivalent abstract entities. Examples of abstract species are natural number and color. A *concrete species* describes the set of attributes of essentially equivalent concrete entities. Examples of concrete species are man and U.S. state.

A *function* is a rule that associates one or more abstract entities, called *arguments*, from corresponding species with an abstract entity, called the *result*, from another species. Examples of functions are the successor function, which associates each natural number with the one that immediately follows it, and the function that associates with two colors the result of blending them.

An *abstract genus* describes different abstract species that are similar in some respect. Examples of abstract genera are number and binary operator. A *concrete genus* describes different concrete species similar in some respect. Examples of concrete genera are mammal and biped.

An entity belongs to a single species, which provides the rules for its construction or existence. An entity can belong to several genera, each of which describes certain properties.

We show later in the chapter that objects and values represent entities, types represent species, and concepts represent genera.

1.2 Values

Unless we know the interpretation, the only things we see in a computer are 0s and 1s. A *datum* is a finite sequence of 0s and 1s.

A *value type* is a correspondence between a species (abstract or concrete) and a set of datums. A datum corresponding to a particular entity is called a *representation* of the entity; the entity is called the *interpretation* of the datum. We refer to a datum together with its interpretation as a *value*. Examples of values are integers represented in 32-bit two's complement big-endian format and rational numbers represented as a concatenation of two 32-bit sequences, interpreted as integer numerator and denominator, represented as two's complement big-endian values.

A datum is *well formed* with respect to a value type if and only if that datum represents an abstract entity. For example, every sequence of 32 bits is well formed when interpreted as a two's-complement integer; an IEEE 754 floating-point NaN (Not a Number) is not well formed when interpreted as a real number.

A value type is *properly partial* if its values represent a proper subset of the abstract entities in the corresponding species; otherwise it is *total*. For example, the type `int` is properly partial, while the type `bool` is total.

A value type is *uniquely represented* if and only if at most one value corresponds to each abstract entity. For example, a type representing a truth value as a byte

that interprets zero as false and nonzero as true is not uniquely represented. A type representing an integer as a sign bit and an unsigned magnitude does not provide a unique representation of zero. A type representing an integer in two's complement is uniquely represented.

A value type is *ambiguous* if and only if a value of the type has more than one interpretation. The negation of ambiguous is *unambiguous*. For example, a type representing a calendar year over a period longer than a single century as two decimal digits is ambiguous.

Two values of a value type are *equal* if and only if they represent the same abstract entity. They are *representationally equal* if and only if their datums are identical sequences of 0s and 1s.

Lemma 1.1 If a value type is uniquely represented, equality implies representational equality.

Lemma 1.2 If a value type is not ambiguous, representational equality implies equality.

If a value type is uniquely represented, we implement equality by testing that both sequences of 0s and 1s are the same. Otherwise we must implement equality in such a way that preserves its consistency with the interpretations of its arguments. Nonunique representations are chosen when testing equality is done less frequently than operations generating new values and when it is possible to make generating new values faster at the cost of making equality slower. For example, two rational numbers represented as pairs of integers are equal if they reduce to the same lowest terms. Two finite sets represented as unsorted sequences are equal if, after sorting and eliminating duplicates, their corresponding elements are equal.

Sometimes, implementing true *behavioral* equality is too expensive or even impossible, as in the case for a type of encodings of computable functions. In these cases we must settle for the weaker *representational* equality: that two values are the same sequence of 0s and 1s.

Computers *implement* functions on abstract entities as functions on values. While values reside in memory, a properly implemented function on values does not depend on particular memory addresses: It implements a mapping from values to values.

A function defined on a value type is *regular* if and only if it respects equality: Substituting an equal value for an argument gives an equal result. Most numeric functions are regular. An example of a numeric function that is not regular is the function that returns the numerator of a rational number represented as a pair of

integers, since $\frac{1}{2} = \frac{2}{4}$, but numerator($\frac{1}{2}$) $\neq$ numerator($\frac{2}{4}$). Regular functions allow *equational reasoning:* substituting equals for equals.

A nonregular function depends on the representation, not just the interpretation, of its argument. When designing the representation for a value type, two tasks go hand in hand: implementing equality and deciding which functions will be regular.

1.3 Objects

A *memory* is a set of words, each with an *address* and a *content*. The addresses are values of a fixed size, called the *address length*. The contents are values of another fixed size, called the *word length*. The content of an address is obtained by a *load* operation. The association of a content with an address is changed by a *store* operation. Examples of memories are bytes in main memory and blocks on a disk drive.

An *object* is a representation of a concrete entity as a value in memory. An object has a *state* that is a value of some value type. The state of an object is changeable. Given an object corresponding to a concrete entity, its state corresponds to a snapshot of that entity. An object owns a set of *resources*, such as memory words or records in a file, to hold its state.

While the value of an object is a contiguous sequence of 0s and 1s, the resources in which these 0s and 1s are stored are not necessarily contiguous. It is the interpretation that gives unity to an object. For example, two `doubles` may be interpreted as a single complex number even if they are not adjacent. The resources of an object might even be in different memories. This book, however, deals only with objects residing in a single memory with one address space. Every object has a unique *starting address*, from which all its resources can be reached.

An *object type* is a pattern for storing and modifying values in memory. Corresponding to every object type is a value type describing states of objects of that type. Every object belongs to an object type. An example of an object type is integers represented in 32-bit two's complement little-endian format aligned to a 4-byte address boundary.

Values and objects play complementary roles. Values are unchanging and are independent of any particular implementation in the computer. Objects are changeable and have computer-specific implementations. The state of an object at any point in time can be described by a value; this value could in principle be written down on paper (making a snapshot) or *serialized* and sent over a communication link.

Describing the states of objects in terms of values allows us to abstract from the particular implementations of the objects when discussing equality. Functional programming deals with values; imperative programming deals with objects.

We use values to represent entities. Since values are unchanging, they can represent abstract entities. Sequences of values can also represent sequences of snapshots of concrete entities. Objects hold values representing entities. Since objects are changeable, they can represent concrete entities by taking on a new value to represent a change in the entity. Objects can also represent abstract entities: staying constant or taking on different approximations to the abstract.

We use objects in the computer for the following three reasons.

1. Objects model changeable concrete entities, such as employee records in a payroll application.
2. Objects provide a powerful way to implement functions on values, such as a procedure implementing the square root of a floating-point number using an iterative algorithm.
3. Computers with memory constitute the only available realization of a universal computational device.

Some properties of value types carry through to object types. An object is *well formed* if and only if its state is well formed. An object type is *properly partial* if and only if its value type is properly partial; otherwise it is *total*. An object type is *uniquely represented* if and only if its value type is uniquely represented.

Since concrete entities have identities, objects representing them need a corresponding notion of identity. An *identity token* is a unique value expressing the identity of an object and is computed from the value of the object and the address of its resources. Examples of identity tokens are the address of the object, an index into an array where the object is stored, and an employee number in a personnel record. Testing equality of identity tokens corresponds to testing identity. During the lifetime of an application, a particular object could use different identity tokens as it moves either within a data structure or from one data structure to another.

Two objects of the same type are *equal* if and only if their states are equal. If two objects are equal, we say that one is a *copy* of the other. Making a change to an object does not affect any copy of it.

This book uses a programming language that has no way to describe values and value types as separate from objects and object types. So from this point on, when we refer to types without qualification, we mean object types.

1.4 Procedures

A *procedure* is a sequence of instructions that modifies the state of some objects; it may also construct or destroy objects.

The objects with which a procedure interacts can be divided into four kinds, corresponding to the intentions of the programmer.

1. *Input/output* consists of objects passed to/from a procedure directly or indirectly through its arguments or returned result.
2. *Local state* consists of objects created, destroyed, and usually modified during a single invocation of the procedure.
3. *Global state* consists of objects accessible to this and other procedures across multiple invocations.
4. *Own state* consists of objects accessible only to this procedure (and its affiliated procedures) but shared across multiple invocations.

An object is passed *directly* if it is passed as an argument or returned as the result and is passed *indirectly* if it is passed via a pointer or pointerlike object. An object is an *input* to a procedure if it is read, but not modified, by the procedure. An object is an *output* from a procedure if it is written, created, or destroyed by the procedure, but its initial state is not read by the procedure. An object is an *input/output* of a procedure if it is modified as well as read by the procedure.

A *computational basis* for a type is a finite set of procedures that enable the construction of any other procedure on the type. A basis is *efficient* if and only if any procedure implemented using it is as efficient as an equivalent procedure written in terms of an alternative basis. For example, a basis for unsigned k-bit integers providing only zero, equality, and the successor function is not efficient, since the complexity of addition in terms of successor is exponential in k.

A basis is *expressive* if and only if it allows compact and convenient definitions of procedures on the type. In particular, all the common mathematical operations need to be provided when they are appropriate. For example, subtraction could be implemented using negation and addition but should be included in an expressive basis. Similarly, negation could be implemented using subtraction and zero but should be included in an expressive basis.

1.5 Regular Types

There is a set of procedures whose inclusion in the computational basis of a type lets us place objects in data structures and use algorithms to copy objects from one data structure to another. We call types having such a basis *regular*, since their