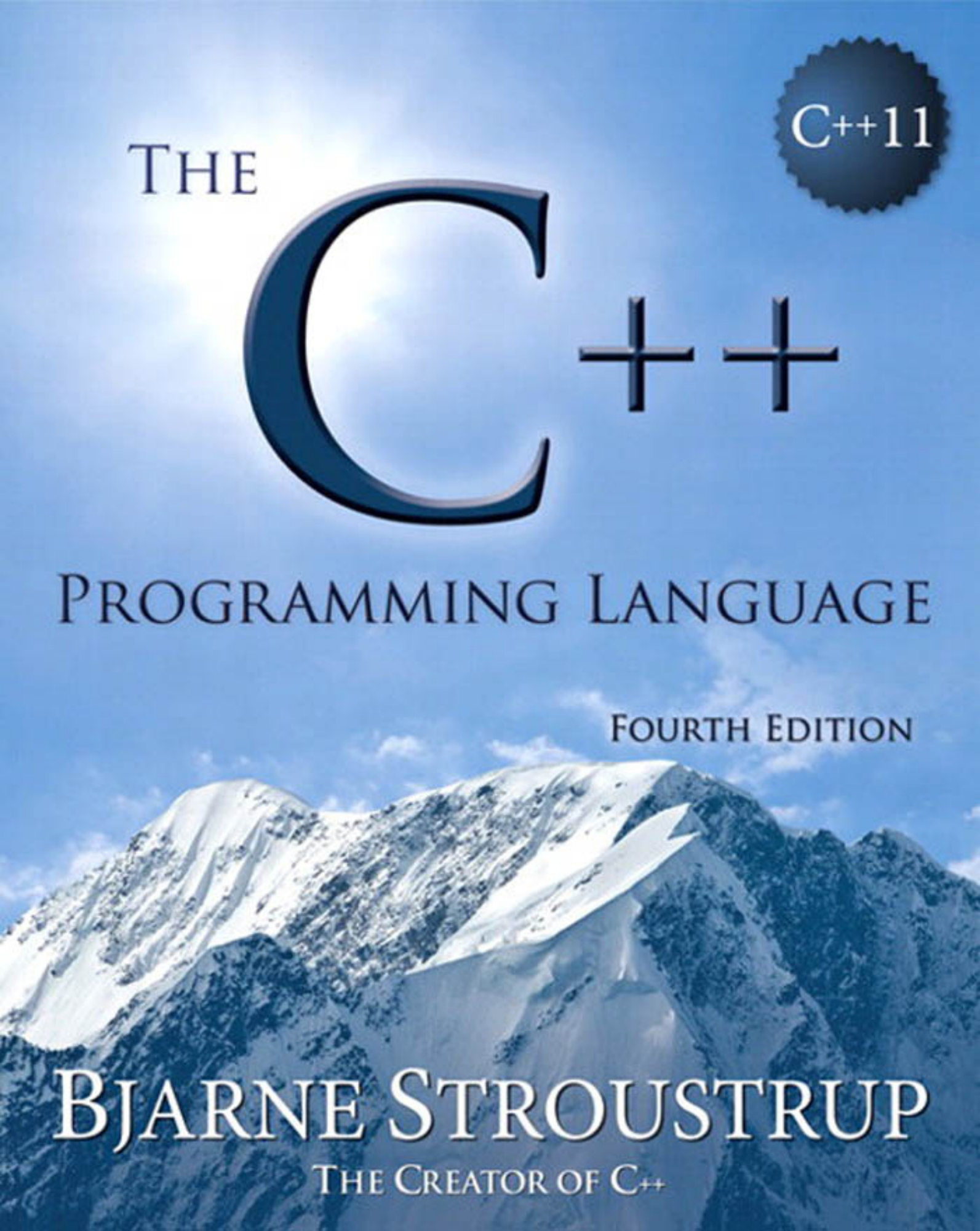
C++11
THE
C++
PROGRAMMING LANGUAGE
FOURTH EDITION
BJARNE STROUSTRUP
THE CREATOR OF C++

The C++ Programming Language

Fourth Edition

Bjarne Stroustrup

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Totonto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

Stroustrup, Bjarne.
The C++ programming language / Bjarne Stroustrup.—Fourth edition.
pages cm
Includes bibliographical references and index.
ISBN 978-0-321-56384-2 (pbk. : alk. paper)—ISBN 0-321-56384-0 (pbk. : alk. paper)
1. C++ (Computer programming language) I. Title.

QA76.73.C153 S77 2013
005.13'3—dc23 2013002159

This book was typeset in Times and Helvetica by the author.

ISBN-13: 978-0-321-56384-2
ISBN-10: 0-321-56384-0
Text printed in the United States on recycled paper at Edwards Brothers Malloy in Ann Arbor, Michigan.
Second printing, June 2013

Contents

Preface

All problems in computer science
can be solved by another level of indirection,
except for the problem of too many layers of indirection.
– David J. Wheeler

C++ feels like a new language. That is, I can express my ideas more clearly, more simply, and more directly in C++11 than I could in C++98. Furthermore, the resulting programs are better checked by the compiler and run faster.

In this book, I aim for *completeness*. I describe every language feature and standard-library component that a professional programmer is likely to need. For each, I provide:

- *Rationale*: What kinds of problems is it designed to help solve? What principles underlie the design? What are the fundamental limitations?
- *Specification*: What is its definition? The level of detail is chosen for the expert programmer; the aspiring language lawyer can follow the many references to the ISO standard.
- *Examples*: How can it be used well by itself and in combination with other features? What are the key techniques and idioms? What are the implications for maintainability and performance?

The use of C++ has changed dramatically over the years and so has the language itself. From the point of view of a programmer, most of the changes have been improvements. The current ISO standard C++ (ISO/IEC 14882-2011, usually called C++11) is simply a far better tool for writing quality software than were previous versions. How is it a better tool? What kinds of programming styles and techniques does modern C++ support? What language and standard-library features support those techniques? What are the basic building blocks of elegant, correct, maintainable, and efficient C++ code? Those are the key questions answered by this book. Many answers are not the same as you would find with 1985, 1995, or 2005 vintage C++: progress happens.

C++ is a general-purpose programming language emphasizing the design and use of type-rich, lightweight abstractions. It is particularly suited for resource-constrained applications, such as those found in software infrastructures. C++ rewards the programmer who takes the time to master

techniques for writing quality code. C++ is a language for someone who takes the task of programming seriously. Our civilization depends critically on software; it had better be quality software.

There are billions of lines of C++ deployed. This puts a premium on stability, so 1985 and 1995 C++ code still works and will continue to work for decades. However, for all applications, you can do better with modern C++; if you stick to older styles, you will be writing lower-quality and worse-performing code. The emphasis on stability also implies that standards-conforming code you write today will still work a couple of decades from now. All code in this book conforms to the 2011 ISO C++ standard.

This book is aimed at three audiences:

- C++ programmers who want to know what the latest ISO C++ standard has to offer,
- C programmers who wonder what C++ provides beyond C, and
- People with a background in application languages, such as Java, C#, Python, and Ruby, looking for something "closer to the machine" – something more flexible, something offering better compile-time checking, or something offering better performance.

Naturally, these three groups are not disjoint – a professional software developer masters more than just one programming language.

This book assumes that its readers are programmers. If you ask, "What's a for-loop?" or "What's a compiler?" then this book is not (yet) for you; instead, I recommend my *Programming: Principles and Practice Using C++* to get started with programming and C++. Furthermore, I assume that readers have some maturity as software developers. If you ask "Why bother testing?" or say, "All languages are basically the same; just show me the syntax" or are confident that there is a single language that is ideal for every task, this is not the book for you.

What features does C++11 offer over and above C++98? A machine model suitable for modern computers with lots of concurrency. Language and standard-library facilities for doing systems-level concurrent programming (e.g., using multicores). Regular expression handling, resource management pointers, random numbers, improved containers (including, hash tables), and more. General and uniform initialization, a simpler **for**-statement, move semantics, basic Unicode support, lambdas, general constant expressions, control over class defaults, variadic templates, user-defined literals, and more. Please remember that those libraries and language features exist to support programming techniques for developing quality software. They are meant to be used in combination – as bricks in a building set – rather than to be used individually in relative isolation to solve a specific problem. A computer is a universal machine, and C++ serves it in that capacity. In particular, C++'s design aims to be sufficiently flexible and general to cope with future problems undreamed of by its designers.

Acknowledgments

In addition to the people mentioned in the acknowledgment sections of the previous editions, I would like to thank Pete Becker, Hans-J. Boehm, Marshall Clow, Jonathan Coe, Lawrence Crowl, Walter Daugherty, J. Daniel Garcia, Robert Harle, Greg Hickman, Howard Hinnant, Brian Kernighan, Daniel Krügler, Nevin Liber, Michel Michaud, Gary Powell, Jan Christiaan van Winkel, and Leor Zolman. Without their help this book would have been much poorer.

Thanks to Howard Hinnant for answering many questions about the standard library.

Andrew Sutton is the author of the Origin library, which was the testbed for much of the discussion of emulating concepts in the template chapters, and of the matrix library that is the topic of Chapter 29. The Origin library is open source and can be found by searching the Web for ''Origin'' and ''Andrew Sutton.''

Thanks to my graduate design class for finding more problems with the ''tour chapters'' than anyone else.

Had I been able to follow every piece of advice of my reviewers, the book would undoubtedly have been much improved, but it would also have been hundreds of pages longer. Every expert reviewer suggested adding technical details, advanced examples, and many useful development conventions; every novice reviewer (or educator) suggested adding examples; and most reviewers observed (correctly) that the book may be too long.

Thanks to Princeton University's Computer Science Department, and especially Prof. Brian Kernighan, for hosting me for part of the sabbatical that gave me time to write this book.

Thanks to Cambridge University's Computer Lab, and especially Prof. Andy Hopper, for hosting me for part of the sabbatical that gave me time to write this book.

Thanks to my editor, Peter Gordon, and his production team at Addison-Wesley for their help and patience.

College Station, Texas *Bjarne Stroustrup*

This page intentionally left blank

Preface to the Third Edition

Programming is understanding.
– Kristen Nygaard

I find using C++ more enjoyable than ever. C++'s support for design and programming has improved dramatically over the years, and lots of new helpful techniques have been developed for its use. However, C++ is not *just* fun. Ordinary practical programmers have achieved significant improvements in productivity, maintainability, flexibility, and quality in projects of just about any kind and scale. By now, C++ has fulfilled most of the hopes I originally had for it, and also succeeded at tasks I hadn't even dreamt of.

This book introduces standard C++† and the key programming and design techniques supported by C++. Standard C++ is a far more powerful and polished language than the version of C++ introduced by the first edition of this book. New language features such as namespaces, exceptions, templates, and run-time type identification allow many techniques to be applied more directly than was possible before, and the standard library allows the programmer to start from a much higher level than the bare language.

About a third of the information in the second edition of this book came from the first. This third edition is the result of a rewrite of even larger magnitude. It offers something to even the most experienced C++ programmer; at the same time, this book is easier for the novice to approach than its predecessors were. The explosion of C++ use and the massive amount of experience accumulated as a result makes this possible.

The definition of an extensive standard library makes a difference to the way C++ concepts can be presented. As before, this book presents C++ independently of any particular implementation, and as before, the tutorial chapters present language constructs and concepts in a "bottom up" order so that a construct is used only after it has been defined. However, it is much easier to use a well-designed library than it is to understand the details of its implementation. Therefore, the standard library can be used to provide realistic and interesting examples well before a reader can be assumed to understand its inner workings. The standard library itself is also a fertile source of programming examples and design techniques.

This book presents every major C++ language feature and the standard library. It is organized around language and library facilities. However, features are presented in the context of their use.

† ISO/IEC 14882, Standard for the C++ Programming Language.

That is, the focus is on the language as the tool for design and programming rather than on the language in itself. This book demonstrates key techniques that make C++ effective and teaches the fundamental concepts necessary for mastery. Except where illustrating technicalities, examples are taken from the domain of systems software. A companion, *The Annotated C++ Language Standard*, presents the complete language definition together with annotations to make it more comprehensible.

The primary aim of this book is to help the reader understand how the facilities offered by C++ support key programming techniques. The aim is to take the reader far beyond the point where he or she gets code running primarily by copying examples and emulating programming styles from other languages. Only a good understanding of the ideas behind the language facilities leads to mastery. Supplemented by implementation documentation, the information provided is sufficient for completing significant real-world projects. The hope is that this book will help the reader gain new insights and become a better programmer and designer.

Acknowledgments

In addition to the people mentioned in the acknowledgement sections of the first and second editions, I would like to thank Matt Austern, Hans Boehm, Don Caldwell, Lawrence Crowl, Alan Feuer, Andrew Forrest, David Gay, Tim Griffin, Peter Juhl, Brian Kernighan, Andrew Koenig, Mike Mowbray, Rob Murray, Lee Nackman, Joseph Newcomer, Alex Stepanov, David Vandevoorde, Peter Weinberger, and Chris Van Wyk for commenting on draft chapters of this third edition. Without their help and suggestions, this book would have been harder to understand, contained more errors, been slightly less complete, and probably been a little bit shorter.

I would also like to thank the volunteers on the C++ standards committees who did an immense amount of constructive work to make C++ what it is today. It is slightly unfair to single out individuals, but it would be even more unfair not to mention anyone, so I'd like to especially mention Mike Ball, Dag Brück, Sean Corfield, Ted Goldstein, Kim Knuttila, Andrew Koenig, Dmitry Lenkov, Nathan Myers, Martin O'Riordan, Tom Plum, Jonathan Shopiro, John Spicer, Jerry Schwarz, Alex Stepanov, and Mike Vilot, as people who each directly cooperated with me over some part of C++ and its standard library.

After the initial printing of this book, many dozens of people have mailed me corrections and suggestions for improvements. I have been able to accommodate many of their suggestions within the framework of the book so that later printings benefitted significantly. Translators of this book into many languages have also provided many clarifications. In response to requests from readers, I have added appendices D and E. Let me take this opportunity to thank a few of those who helped: Dave Abrahams, Matt Austern, Jan Bielawski, Janina Mincer Daszkiewicz, Andrew Koenig, Dietmar Kühl, Nicolai Josuttis, Nathan Myers, Paul E. Sevinç, Andy Tenne-Sens, Shoichi Uchida, Ping-Fai (Mike) Yang, and Dennis Yelle.

Murray Hill, New Jersey *Bjarne Stroustrup*

Preface to the Second Edition

The road goes ever on and on.
– Bilbo Baggins

As promised in the first edition of this book, C++ has been evolving to meet the needs of its users. This evolution has been guided by the experience of users of widely varying backgrounds working in a great range of application areas. The C++ user-community has grown a hundredfold during the six years since the first edition of this book; many lessons have been learned, and many techniques have been discovered and/or validated by experience. Some of these experiences are reflected here.

The primary aim of the language extensions made in the last six years has been to enhance C++ as a language for data abstraction and object-oriented programming in general and to enhance it as a tool for writing high-quality libraries of user-defined types in particular. A "high-quality library," is a library that provides a concept to a user in the form of one or more classes that are convenient, safe, and efficient to use. In this context, *safe* means that a class provides a specific type-safe interface between the users of the library and its providers; *efficient* means that use of the class does not impose significant overheads in run-time or space on the user compared with hand-written C code.

This book presents the complete C++ language. Chapters 1 through 10 give a tutorial introduction; Chapters 11 through 13 provide a discussion of design and software development issues; and, finally, the complete C++ reference manual is included. Naturally, the features added and resolutions made since the original edition are integral parts of the presentation. They include refined overloading resolution, memory management facilities, and access control mechanisms, type-safe linkage, **const** and **static** member functions, abstract classes, multiple inheritance, templates, and exception handling.

C++ is a general-purpose programming language; its core application domain is systems programming in the broadest sense. In addition, C++ is successfully used in many application areas that are not covered by this label. Implementations of C++ exist from some of the most modest microcomputers to the largest supercomputers and for almost all operating systems. Consequently, this book describes the C++ language itself without trying to explain a particular implementation, programming environment, or library.

This book presents many examples of classes that, though useful, should be classified as "toys." This style of exposition allows general principles and useful techniques to stand out more clearly than they would in a fully elaborated program, where they would be buried in details. Most

of the useful classes presented here, such as linked lists, arrays, character strings, matrices, graphics classes, associative arrays, etc., are available in ''bulletproof'' and/or ''goldplated'' versions from a wide variety of commercial and non-commercial sources. Many of these ''industrial strength'' classes and libraries are actually direct and indirect descendants of the toy versions found here.

This edition provides a greater emphasis on tutorial aspects than did the first edition of this book. However, the presentation is still aimed squarely at experienced programmers and endeavors not to insult their intelligence or experience. The discussion of design issues has been greatly expanded to reflect the demand for information beyond the description of language features and their immediate use. Technical detail and precision have also been increased. The reference manual, in particular, represents many years of work in this direction. The intent has been to provide a book with a depth sufficient to make more than one reading rewarding to most programmers. In other words, this book presents the C++ language, its fundamental principles, and the key techniques needed to apply it. Enjoy!

Acknowledgments

In addition to the people mentioned in the acknowledgements section in the preface to the first edition, I would like to thank Al Aho, Steve Buroff, Jim Coplien, Ted Goldstein, Tony Hansen, Lorraine Juhl, Peter Juhl, Brian Kernighan, Andrew Koenig, Bill Leggett, Warren Montgomery, Mike Mowbray, Rob Murray, Jonathan Shopiro, Mike Vilot, and Peter Weinberger for commenting on draft chapters of this second edition. Many people influenced the development of C++ from 1985 to 1991. I can mention only a few: Andrew Koenig, Brian Kernighan, Doug McIlroy, and Jonathan Shopiro. Also thanks to the many participants of the ''external reviews'' of the reference manual drafts and to the people who suffered through the first year of X3J16.

Murray Hill, New Jersey *Bjarne Stroustrup*

Preface to the First Edition

Language shapes the way we think,
and determines what we can think about.
– B.L.Whorf

C++ is a general purpose programming language designed to make programming more enjoyable for the serious programmer. Except for minor details, C++ is a superset of the C programming language. In addition to the facilities provided by C, C++ provides flexible and efficient facilities for defining new types. A programmer can partition an application into manageable pieces by defining new types that closely match the concepts of the application. This technique for program construction is often called *data abstraction.* Objects of some user-defined types contain type information. Such objects can be used conveniently and safely in contexts in which their type cannot be determined at compile time. Programs using objects of such types are often called *object based*. When used well, these techniques result in shorter, easier to understand, and easier to maintain programs.

The key concept in C++ is *class*. A class is a user-defined type. Classes provide data hiding, guaranteed initialization of data, implicit type conversion for user-defined types, dynamic typing, user-controlled memory management, and mechanisms for overloading operators. C++ provides much better facilities for type checking and for expressing modularity than C does. It also contains improvements that are not directly related to classes, including symbolic constants, inline substitution of functions, default function arguments, overloaded function names, free store management operators, and a reference type. C++ retains C's ability to deal efficiently with the fundamental objects of the hardware (bits, bytes, words, addresses, etc.). This allows the user-defined types to be implemented with a pleasing degree of efficiency.

C++ and its standard libraries are designed for portability. The current implementation will run on most systems that support C. C libraries can be used from a C++ program, and most tools that support programming in C can be used with C++.

This book is primarily intended to help serious programmers learn the language and use it for nontrivial projects. It provides a complete description of C++, many complete examples, and many more program fragments.

Acknowledgments

C++ could never have matured without the constant use, suggestions, and constructive criticism of many friends and colleagues. In particular, Tom Cargill, Jim Coplien, Stu Feldman, Sandy Fraser, Steve Johnson, Brian Kernighan, Bart Locanthi, Doug McIlroy, Dennis Ritchie, Larry Rosler, Jerry Schwarz, and Jon Shopiro provided important ideas for development of the language. Dave Presotto wrote the current implementation of the stream I/O library.

In addition, hundreds of people contributed to the development of C++ and its compiler by sending me suggestions for improvements, descriptions of problems they had encountered, and compiler errors. I can mention only a few: Gary Bishop, Andrew Hume, Tom Karzes, Victor Milenkovic, Rob Murray, Leonie Rose, Brian Schmult, and Gary Walker.

Many people have also helped with the production of this book, in particular, Jon Bentley, Laura Eaves, Brian Kernighan, Ted Kowalski, Steve Mahaney, Jon Shopiro, and the participants in the C++ course held at Bell Labs, Columbus, Ohio, June 26-27, 1985.

Murray Hill, New Jersey *Bjarne Stroustrup*

Part I

Introduction

This introduction gives an overview of the major concepts and features of the C++ programming language and its standard library. It also provides an overview of this book and explains the approach taken to the description of the language facilities and their use. In addition, the introductory chapters present some background information about C++, the design of C++, and the use of C++.

Chapters

"... and you, Marcus, you have given me many things; now I shall give you this good advice. Be many people. Give up the game of being always Marcus Cocoza. You have worried too much about Marcus Cocoza, so that you have been really his slave and prisoner. You have not done anything without first considering how it would affect Marcus Cocoza's happiness and prestige. You were always much afraid that Marcus might do a stupid thing, or be bored. What would it really have mattered? All over the world people are doing stupid things ... I should like you to be easy, your little heart to be light again. You must from now, be more than one, many people, as many as you can think of ..."

– Karen Blixen,
The Dreamers from *Seven Gothic Tales* (1934)

1

Notes to the Reader

Hurry Slowly
(festina lente).
– Octavius, Caesar Augustus

- The Structure of This Book
  Introduction; Basic Facilities; Abstraction Mechanisms; The Standard Library; Examples and References
- The Design of C++
  Programming Styles; Type Checking; C Compatibility; Language, Libraries, and Systems
- Learning C++
  Programming in C++; Suggestions for C++ Programmers; Suggestions for C Programmers; Suggestions for Java Programmers
- History
  Timeline; The Early Years; The 1998 Standard; The 2011 Standard; What is C++ Used for?
- Advice
- References

1.1 The Structure of This Book

A pure tutorial sorts its topics so that no concept is used before it has been introduced; it must be read linearly starting with page one. Conversely, a pure reference manual can be accessed starting at any point; it describes each topic succinctly with references (forward and backward) to related topics. A pure tutorial can in principle be read without prerequisites – it carefully describes all. A pure reference can be used only by someone familiar with all fundamental concepts and techniques. This book combines aspects of both. If you know most concepts and techniques, you can access it on a per-chapter or even on a per-section basis. If not, you can start at the beginning, but try not to get bogged down in details. Use the index and the cross-references.

Making parts of the book relatively self-contained implies some repetition, but repetition also serves as review for people reading the book linearly. The book is heavily cross-referenced both to itself and to the ISO C++ standard. Experienced programmers can read the (relatively) quick ''tour'' of C++ to gain the overview needed to use the book as a reference. This book consists of four parts:

Part I *Introduction*: Chapter 1 (this chapter) is a guide to this book and provides a bit of C++ background. Chapters 2-5 give a quick introduction to the C++ language and its standard library.

Part II *Basic Facilities*: Chapters 6-15 describe C++'s built-in types and the basic facilities for constructing programs out of them.

Part III *Abstraction Mechanisms*: Chapters 16-29 describe C++'s abstraction mechanisms and their use for object-oriented and generic programming.

Part IV Chapters 30-44 provide an overview of the standard library and a discussion of compatibility issues.

1.1.1 Introduction

This chapter, Chapter 1, provides an overview of this book, some hints about how to use it, and some background information about C++ and its use. You are encouraged to skim through it, read what appears interesting, and return to it after reading other parts of the book. Please do not feel obliged to read it all carefully before proceeding.

The following chapters provide an overview of the major concepts and features of the C++ programming language and its standard library:

Chapter 2 *A Tour of C++: The Basics* describes C++'s model of memory, computation, and error handling.

Chapter 3 *A Tour of C++: Abstraction Mechanisms* presents the language features supporting data abstraction, object-oriented programming, and generic programming.

Chapter 4 *A Tour of C++: Containers and Algorithms* introduces strings, simple I/O, containers, and algorithms as provided by the standard library.

Chapter 5 *A Tour of C++: Concurrency and Utilities* outlines the standard-library utilities related to resource management, concurrency, mathematical computation, regular expressions, and more.

This whirlwind tour of C++'s facilities aims to give the reader a taste of what C++ offers. In particular, it should convince readers that C++ has come a long way since the first, second, and third editions of this book.

1.1.2 Basic Facilities

Part II focuses on the subset of C++ that supports the styles of programming traditionally done in C and similar languages. It introduces the notions of type, object, scope, and storage. It presents the fundamentals of computation: expressions, statements, and functions. Modularity – as supported by namespaces, source files, and exception handling – is also discussed:

Chapter 6 *Types and Declarations*: Fundamental types, naming, scopes, initialization, simple type deduction, object lifetimes, and type aliases

Chapter 7 *Pointers, Arrays, and References*
Chapter 8 *Structures, Unions, and Enumerations*
Chapter 9 *Statements*: Declarations as statements, selection statements (**if** and **switch**), iteration statements (**for**, **while**, and **do**), **goto**, and comments
Chapter 10 *Expressions*: A desk calculator example, survey of operators, constant expressions, and implicit type conversion.
Chapter 11 *Select Operations*: Logical operators, the conditional expression, increment and decrement, free store (**new** and **delete**), **{}**-lists, lambda expressions, and explicit type conversion (**static_cast** and **const_cast**)
Chapter 12 *Functions*: Function declarations and definitions, **inline** functions, **constexpr** functions, argument passing, overloaded functions, pre- and postconditions, pointers to functions, and macros
Chapter 13 *Exception Handling*: Styles of error handling, exception guarantees, resource management, enforcing invariants, **throw** and **catch**, a **vector** implementation
Chapter 14 *Namespaces*: **namespace**, modularization and interface, composition using namespaces
Chapter 15 *Source Files and Programs*: Separate compilation, linkage, using header files, and program start and termination

I assume that you are familiar with most of the programming concepts used in Part I. For example, I explain the C++ facilities for expressing recursion and iteration, but I do not go into technical details or spend much time explaining how these concepts are useful.

The exception to this rule is exceptions. Many programmers lack experience with exceptions or got their experience from languages (such as Java) where resource management and exception handling are not integrated. Consequently, the chapter on exception handling (Chapter 13) presents the basic philosophy of C++ exception handling and resource management. It goes into some detail about strategy with a focus on the "Resource Acquisition Is Initialization" technique (RAII).

1.1.3 Abstraction Mechanisms

Part III describes the C++ facilities supporting various forms of abstraction, including object-oriented and generic programming. The chapters fall into three rough categories: classes, class hierarchies, and templates.

The first four chapters concentrate of the classes themselves:

Chapter 16 *Classes*: The notion of a user-defined type, a class, is the foundation of all C++ abstraction mechanisms.
Chapter 17 *Construction, Cleanup, Copy, and Move* shows how a programmer can define the meaning of creation and initialization of objects of a class. Further, the meaning of copy, move, and destruction can be specified.
Chapter 18 *Operator Overloading* presents the rules for giving meaning to operators for user-defined types with an emphasis on conventional arithmetic and logical operators, such as **+**, *, and **&**.
Chapter 19 *Special Operators* discusses the use of user-defined operator for non-arithmetic purposes, such as **[]** for subscripting, **()** for function objects, and **->** for "smart pointers."

Classes can be organized into hierarchies:

Chapter 20 *Derived Classes* presents the basic language facilities for building hierarchies out of classes and the fundamental ways of using them. We can provide complete separation between an interface (an abstract class) and its implementations (derived classes); the connection between them is provided by virtual functions. The C++ model for access control (**public**, **protected**, and **private**) is presented.

Chapter 21 *Class Hierarchies* discusses ways of using class hierarchies effectively. It also presents the notion of multiple inheritance, that is, a class having more than one direct base class.

Chapter 22 *Run-Time Type Information* presents ways to navigate class hierarchies using data stored in objects. We can use **dynamic_cast** to inquire whether an object of a base class was defined as an object of a derived class and use the **typeid** to gain minimal information from an object (such as the name of its class).

Many of the most flexible, efficient, and useful abstractions involve the parameterization of types (classes) and algorithms (functions) with other types and algorithms:

Chapter 23 *Templates* presents the basic principles behind templates and their use. Class templates, function templates, and template aliases are presented.

Chapter 24 *Generic Programming* introduces the basic techniques for designing generic programs. The technique of *lifting* an abstract algorithm from a number of concrete code examples is central, as is the notion of *concepts* specifying a generic algorithm's requirements on its arguments.

Chapter 25 *Specialization* describes how templates are used to generate classes and functions, *specializations*, given a set of template arguments.

Chapter 26 *Instantiation* focuses on the rules for name binding.

Chapter 27 *Templates and Hierarchies* explains how templates and class hierarchies can be used in combination.

Chapter 28 *Metaprogramming* explores how templates can be used to generate programs. Templates provide a Turing-complete mechanism for generating code.

Chapter 29 *A Matrix Design* gives a longish example to show how language features can be used in combination to solve a complex design problem: the design of an N-dimensional matrix with near-arbitrary element types.

The language features supporting abstraction techniques are described in the context of those techniques. The presentation technique in Part III differs from that of Part II in that I don't assume that the reader knows the techniques described.

1.1.4 The Standard Library

The library chapters are less tutorial than the language chapters. In particular, they are meant to be read in any order and can be used as a user-level manual for the library components:

Chapter 30 *Standard-Library Overview* gives an overview of the standard library, lists the standard-library headers, and presents language support and diagnostics support, such as **exception** and **system_error**.

Chapter 31 *STL Containers* presents the containers from the iterators, containers, and algorithms framework (called *the STL*), including **vector**, **map**, and **unordered_set**.

Chapter 32 *STL Algorithms* presents the algorithms from the STL, including **find()**, **sort()**, and **merge()**.

Chapter 33 *STL Iterators* presents iterators and other utilities from the STL, including **reverse_iterator**, **move_iterator**, and **function**.

Chapter 34 *Memory and Resources* presents utility components related to memory and resource management, such as **array**, **bitset**, **pair**, **tuple**, **unique_ptr**, **shared_ptr**, allocators, and the garbage collector interface.

Chapter 35 *Utilities* presents minor utility components, such as time utilities, type traits, and various type functions.

Chapter 36 *Strings* documents the **string** library, including the character traits that are the basis for the use of different character sets.

Chapter 37 *Regular Expressions* describes the regular expression syntax and the various ways of using it for string matching, including **regex_match()** for matching a complete string, **regex_search()** for finding a pattern in a string, **regex_replace()** for simple replacement, and **regex_iterator** for general traversal of a stream of characters.

Chapter 38 *I/O Streams* documents the stream I/O library. It describes formatted and unformatted input and output, error handling, and buffering.

Chapter 39 *Locales* describes class **locale** and its various **facet**s that provide support for the handling of cultural differences in character sets, formatting of numeric values, formatting of date and time, and more.

Chapter 40 *Numerics* describes facilities for numerical computation (such as **complex**, **valarray**, random numbers, and generalized numerical algorithms).

Chapter 41 *Concurrency* presents the C++ basic memory model and the facilities offered for concurrent programming without locks.

Chapter 42 *Threads and Tasks* presents the classes providing threads-and-locks-style concurrent programming (such as **thread**, **timed_mutex**, **lock_guard**, and **try_lock()**) and the support for task-based concurrency (such as **future** and **async()**).

Chapter 43 *The C Standard Library* documents the C standard library (including **printf()** and **clock()**) as incorporated into the C++ standard library.

Chapter 44 *Compatibility* discusses the relation between C and C++ and between Standard C++ (also called ISO C++) and the versions of C++ that preceded it.

1.1.5 Examples and References

This book emphasizes program organization rather than the design of algorithms. Consequently, I avoid clever or harder-to-understand algorithms. A trivial algorithm is typically better suited to illustrate an aspect of the language definition or a point about program structure. For example, I use a Shell sort where, in real code, a quicksort would be better. Often, reimplementation with a more suitable algorithm is an exercise. In real code, a call of a library function is typically more appropriate than the code used here to illustrate language features.

Textbook examples necessarily give a warped view of software development. By clarifying and simplifying the examples, the complexities that arise from scale disappear. I see no substitute for writing realistically sized programs in order to get an impression of what programming and a

programming language are really like. This book concentrates on the language features and the standard-library facilities. These are the basic techniques from which every program is composed. The rules and techniques for such composition are emphasized.

The selection of examples reflects my background in compilers, foundation libraries, and simulations. The emphasis reflects my interest in systems programming. Examples are simplified versions of what is found in real code. The simplification is necessary to keep programming language and design points from getting lost in details. My ideal is the shortest and clearest example that illustrates a design principle, a programming technique, a language construct, or a library feature. There are no "cute" examples without counterparts in real code. For purely language-technical examples, I use variables named `x` and `y`, types called `A` and `B`, and functions called `f()` and `g()`.

Where possible, the C++ language and library features are presented in the context of their use rather than in the dry manner of a manual. The language features presented and the detail in which they are described roughly reflect my view of what is needed for effective use of C++. The purpose is to give you an idea of how a feature can be used, often in combination with other features. An understanding of every language-technical detail of a language feature or library component is neither necessary nor sufficient for writing good programs. In fact, an obsession with understanding every little detail is a prescription for awful – overelaborate and overly clever – code. What is needed is an understanding of design and programming techniques together with an appreciation of application domains.

I assume that you have access to online information sources. The final arbiter of language and standard-library rules is the ISO C++ standard [C++,2011].

References to parts of this book are of the form §2.3.4 (Chapter 2, section 3, subsection 4) and §iso.5.3.1 (ISO C++ standard, §5.3.1). Italics are used sparingly for emphasis (e.g., "a string literal is *not* acceptable"), for first occurrences of important concepts (e.g., *polymorphism*), and for comments in code examples.

To save a few trees and to simplify additions, the hundreds of exercises for this book have been moved to the Web. Look for them at www.stroustrup.com.

The language and library used in this book are "pure C++" as defined by the C++ standard [C++,2011]. Therefore, the examples should run on every up-to-date C++ implementation. The major program fragments in this book were tried using several C++ implementations. Examples using features only recently adopted into C++ didn't compile on every implementation. However, I see no point in mentioning which implementations failed to compile which examples. Such information would soon be out of date because implementers are working hard to ensure that their implementations correctly accept every C++ feature. See Chapter 44 for suggestions on how to cope with older C++ compilers and with code written for C compilers.

I use C++11 features freely wherever I find them most appropriate. For example, I prefer `{}`-style initializers and `using` for type aliases. In places, that usage may startle "old timers." However, being startled is often a good way to start reviewing material. On the other hand, I don't use new features just because they are new; my ideal is the most elegant expression of the fundamental ideas – and that may very well be using something that has been in C++ or even in C for ages.

Obviously, if you have to use a pre-C++11 compiler (say, because some of your customers have not yet upgraded to the current standard), you have to refrain from using novel features. However, please don't assume that "the old ways" are better or simpler just because they are old and familiar. §44.2 summarizes the differences between C++98 and C++11.

1.2 The Design of C++

The purpose of a programming language is to help express ideas in code. In that, a programming language performs two related tasks: it provides a vehicle for the programmer to specify actions to be executed by the machine, and it provides a set of concepts for the programmer to use when thinking about what can be done. The first purpose ideally requires a language that is ''close to the machine'' so that all important aspects of a machine are handled simply and efficiently in a way that is reasonably obvious to the programmer. The C language was primarily designed with this in mind. The second purpose ideally requires a language that is ''close to the problem to be solved'' so that the concepts of a solution can be expressed directly and concisely. The facilities added to C to create C++, such as function argument checking, **const**, classes, constructors and destructors, exceptions, and templates, were primarily designed with this in mind. Thus, C++ is based on the idea of providing both

- *direct mappings of built-in operations and types to hardware* to provide efficient memory use and efficient low-level operations, and
- *affordable and flexible abstraction mechanisms* to provide user-defined types with the same notational support, range of uses, and performance as built-in types.

This was initially achieved by applying ideas from Simula to C. Over the years, further application of these simple ideals resulted in a far more general, efficient, and flexible set of facilities. The result supports a synthesis of programming styles that can be simultaneously *efficient* and *elegant*.

The design of C++ has focused on programming techniques dealing with fundamental notions such as memory, mutability, abstraction, resource management, expression of algorithms, error handling, and modularity. Those are the most important concerns of a systems programmer and more generally of programmers of resource-constrained and high-performance systems.

By defining libraries of classes, class hierarchies, and templates, you can write C++ programs at a much higher level than the one presented in this book. For example, C++ is widely used in financial systems, for game development, and for scientific computation (§1.4.5). For high-level applications programming to be effective and convenient, we need libraries. Using just the bare language features makes almost all programming quite painful. That's true for every general-purpose language. Conversely, given suitable libraries just about any programming task can be pleasant.

My standard introduction of C++ used to start:

- *C++ is a general-purpose programming language with a bias toward systems programming.*

This is still true. What has changed over the years is an increase in the importance, power, and flexibility of C++'s abstraction mechanisms:

- *C++ is a general-purpose programming language providing a direct and efficient model of hardware combined with facilities for defining lightweight abstractions.*

Or terser:

- *C++ is a language for developing and using elegant and efficient abstractions.*

By *general-purpose programming language* I mean a language designed to support a wide variety of uses. C++ has indeed been used for an incredible variety of uses (from microcontrollers to huge distributed commercial applications), but the key point is that C++ is not deliberately specialized for any given application area. No language is ideal for every application and every programmer, but the ideal for C++ is to support the widest possible range of application areas well.

By *systems programming* I mean writing code that directly uses hardware resources, has serious resource constraints, or closely interacts with code that does. In particular, the implementation of software infrastructure (e.g., device drivers, communications stacks, virtual machines, operating systems, operations systems, programming environments, and foundation libraries) is mostly systems programming. The importance of the "bias toward systems programming" qualification in my long-standing characterization of C++ is that C++ has not been simplified (compromised) by ejecting the facilities aimed at the expert-level use of hardware and systems resources in the hope of making it more suitable for other application areas.

Of course, you can also program in ways that completely hide hardware, use expensive abstractions (e.g., every object on the free store and every operation a virtual function), use inelegant styles (e.g., overabstraction), or use essentially no abstractions ("glorified assembly code"). However, many languages can do that, so those are not distinguishing characteristics of C++.

The *Design and Evolution of C++* book [Stroustrup,1994] (known as *D&E*) outlines the ideas and design aims of C++ in greater detail, but two principles should be noted:

- *Leave no room for a lower-level language below C++* (except for assembly code in rare cases). If you can write more efficient code in a lower-level language then that language will most likely become the systems programming language of choice.
- *What you don't use you don't pay for*. If programmers can hand-write reasonable code to simulate a language feature or a fundamental abstraction and provide even slightly better performance, someone will do so, and many will imitate. Therefore, a language feature and a fundamental abstraction must be designed not to waste a single byte or a single processor cycle compared to equivalent alternatives. This is known as *the zero-overhead principle.*

These are Draconian principles, but essential in some (but obviously not all) contexts. In particular, the zero-overhead principle repeatedly led C++ to simpler, more elegant, and more powerful facilities than were first envisioned. The STL is an example (§4.1.1, §4.4, §4.5, Chapter 31, Chapter 32, Chapter 33). These principles have been essential in the effort to raise the level of programming.

1.2.1 Programming Style

Languages features exist to provide support for programming styles. Please don't look at an individual language feature as a solution, but as one building brick from a varied set which can be combined to express solutions.

The general ideals for design and programming can be expressed simply:

- Express ideas directly in code.
- Express independent ideas independently in code.
- Represent relationships among ideas directly in code.
- Combine ideas expressed in code freely – where and only where combinations make sense.
- Express simple ideas simply.

These are ideals shared by many people, but languages designed to support them can differ dramatically. A fundamental reason for that is that a language embodies a set of engineering tradeoffs reflecting differing needs, tastes, and histories of various individuals and communities. C++'s answers to the general design challenges were shaped by its origins in systems programming (going back to C and BCPL [Richards,1980]), its aim to address issues of program complexity through abstraction (going back to Simula), and its history.

The C++ language features most directly support four programming styles:

- Procedural programming
- Data abstraction
- Object-oriented programming
- Generic programming

However, the emphasis is on the support of effective combinations of those. The best (most maintainable, most readable, smallest, fastest, etc.) solution to most nontrivial problems tends to be one that combines aspects of these styles.

As is usual with important terms in the computing world, a wide variety of definitions of these terms are popular in various parts of the computing industry and academia. For example, what I refer to as a "programming style," others call a "programming technique" or a "paradigm." I prefer to use "programming technique" for something more limited and language-specific. I feel uncomfortable with the word "paradigm" as pretentious and (from Kuhn's original definition) having implied claims of exclusivity.

My ideal is language facilities that can be used elegantly in combination to support a continuum of programming styles and a wide variety of programming techniques.

- *Procedural programming*: This is programming focused on processing and the design of suitable data structures. It is what C was designed to support (and Algol, and Fortran, as well as many other languages). C++'s support comes in the form of the built-in types, operators, statements, functions, **struct**s, **union**s, etc. With minor exceptions, C is a subset of C++. Compared to C, C++ provides further support for procedural programming in the form of many additional language constructs and a stricter, more flexible, and more supportive type system.
- *Data abstraction*: This is programming focused on the design of interfaces, hiding implementation details in general and representations in particular. C++ supports concrete and abstract classes. The facilities for defining classes with private implementation details, constructors and destructors, and associated operations directly support this. The notion of an abstract class provides direct support for complete data hiding.
- *Object-oriented programming*: This is programming focused on the design, implementation, and use of class hierarchies. In addition to allowing the definition lattices of classes, C++ provides a variety of features for navigating class lattices and for simplifying the definition of a class out of existing ones. Class hierarchies provide run-time polymorphism (§20.3.2, §21.2) and encapsulation (§20.4, §20.5).
- *Generic programming*: This is programming focused on the design, implementation, and use of general algorithms. Here, "general" means that an algorithm can be designed to accept a wide variety of types as long as they meet the algorithm's requirements on its arguments. The template is C++'s main support for generic programming. Templates provide (compile-time) parametric polymorphism.

Just about anything that increases the flexibility or efficiency of classes improves the support of all of those styles. Thus, C++ could be (and has been) called *class oriented*.

Each of these styles of design and programming has contributed to the synthesis that is C++. Focusing exclusively on one of these styles is a mistake: except for toy examples, doing so leads to wasted development effort and suboptimal (inflexible, verbose, poorly performing, unmaintainable, etc.) code.

I wince when someone characterizes C++ exclusively through one of these styles (e.g., "C++ is an object-oriented language") or uses a term (e.g., "hybrid" or "mixed paradigm") to imply that a more restrictive language would be preferable. The former misses the fact that all the styles mentioned have contributed something significant to the synthesis; the latter denies the validity of the synthesis. The styles mentioned are not distinct alternatives: each contributes techniques to a more expressive and effective style of programming, and C++ provides direct language support for their use in combination.

From its inception, the design of C++ aimed at a synthesis of programming and design styles. Even the earliest published account of C++ [Stroustrup,1982] presents examples that use these different styles in combination and presents language features aimed at supporting such combinations:

- *Classes* support all of the mentioned styles; all rely on the user representing ideas as user-defined types or objects of user-defined types.
- *Public/private access control* supports data abstraction and object-oriented programming by making a clear distinction between interface and implementation.
- *Member functions, constructors, destructors, and user-defined assignment* provide a clean functional interface to objects as needed by data abstraction and object-oriented programming. They also provide a uniform notation as needed for generic programming. More general overloading had to wait until 1984 and uniform initialization until 2010.
- *Function declarations* provide specific statically checked interfaces to member functions as well as freestanding functions, so they support all of the mentioned styles. They are necessary for overloading. At the time, C lacked "function prototypes" but Simula had function declarations as well as member functions.
- *Generic functions and parameterized types* (generated from functions and classes using macros) support generic programming. Templates had to wait until 1988.
- *Base and derived classes* provide the foundation for object-oriented programming and some forms of data abstraction. Virtual functions had to wait until 1983.
- *Inlining* made the use of these facilities affordable in systems programming and for building run-time and space efficient libraries.

These early features are general abstraction mechanisms, rather than support for disjoint programming styles. Today's C++ provides much better support for design and programming based on lightweight abstraction, but the aim of elegant and efficient code was there from the very beginning. The developments since 1981 provide much better support for the synthesis of the programming styles ("paradigms") originally considered and significantly improve their integration.

The fundamental object in C++ has identity; that is, it is located in a specific location in memory and can be distinguished from other objects with (potentially) the same value by comparing addresses. Expressions denoting such objects are called *lvalues* (§6.4). However, even from the earliest days of C++'s ancestors [Barron,1963] there have also been objects without identity (objects for which an address cannot be safely stored for later use). In C++11, this notion of *rvalue* has been developed into a notion of a value that can be moved around cheaply (§3.3.2, §6.4.1, §7.7.2). Such objects are the basis of techniques that resemble what is found in functional programming (where the notion of objects with identity is viewed with horror). This nicely complements the techniques and language features (e.g., lambda expressions) developed primarily for generic programming. It also solves classical problems related to "simple abstract data types," such as how to elegantly and efficiently return a large matrix from an operation (e.g., a matrix `+`).

From the very earliest days, C++ programs and the design of C++ itself have been concerned about resource management. The ideal was (and is) for resource management to be

- simple (for implementers and especially for users),
- general (a resource is anything that has to be acquired from somewhere and later released),
- efficient (obey the zero-overhead principle; §1.2),
- perfect (no leaks are acceptable), and
- statically type-safe.

Many important C++ classes, such as the standard library's **vector**, **string**, **thread**, **mutex**, **unique_ptr**, **fstream**, and **regex**, are resource handles. Foundation and application libraries beyond the standard provided many more examples, such as Matrix and Widget. The initial step in supporting the notion of resource handles was taken with the provision of constructors and destructors in the very first "C with Classes" draft. This was soon backed with the ability to control copy by defining assignment as well as copy constructors. The introduction of move constructors and move assignments (§3.3) in C++11 completes this line of thinking by allowing cheap movement of potentially large objects from scope to scope (§3.3.2) and to simply control the lifetime of polymorphic or shared objects (§5.2.1).

The facilities supporting resource management also benefit abstractions that are not resource handles. Any class that establishes and maintains an invariant relies on a subset of those features.

1.2.2 Type Checking

The connection between the language in which we think/program and the problems and solutions we can imagine is very close. For this reason, restricting language features with the intent of eliminating programmer errors is, at best, dangerous. A language provides a programmer with a set of conceptual tools; if these are inadequate for a task, they will be ignored. Good design and the absence of errors cannot be guaranteed merely by the presence or absence of specific language features. However, the language features and the type system are provided for the programmer to precisely and concisely represent a design in code.

The notion of static types and compile-time type checking is central to effective use of C++. The use of static types is key to expressiveness, maintainability, and performance. Following Simula, the design of user-defined types with interfaces that are checked at compile time is key to the expressiveness of C++. The C++ type system is extensible in nontrivial ways (Chapter 3, Chapter 16, Chapter 18, Chapter 19, Chapter 21, Chapter 23, Chapter 28, Chapter 29), aiming for equal support for built-in types and user-defined types.

C++ type-checking and data-hiding features rely on compile-time analysis of programs to prevent accidental corruption of data. They do not provide secrecy or protection against someone who is deliberately breaking the rules: C++ protects against accident, not against fraud. They can, however, be used freely without incurring run-time or space overheads. The idea is that to be useful, a language feature must not only be elegant, it must also be affordable in the context of a real-world program.

C++'s static type system is flexible, and the use of simple user-defined types implies little, if any overhead. The aim is to support a style of programming that represents distinct ideas as distinct types, rather than just using generalizations, such as integer, floating-point number, string, "raw memory," and "object," everywhere. A type-rich style of programming makes code more

readable, maintainable, and analyzable. A trivial type system allows only trivial analysis, whereas a type-rich style of programming opens opportunities for nontrivial error detection and optimization. C++ compilers and development tools support such type-based analysis [Stroustrup,2012].

Maintaining most of C as a subset and preserving the direct mapping to hardware needed for the most demanding low-level systems programming tasks implies the ability to break the static type system. However, my ideal is (and always was) complete type safety. In this, I agree with Dennis Ritchie, who said, "C is a strongly typed, weakly checked language." Note that Simula was both type-safe and flexible. In fact, my ideal when I started on C++ was "Algol68 with Classes" rather than "C with Classes." However, the list of solid reasons against basing my work on type-safe Algol68 [Woodward,1974] was long and painful. So, perfect type safety is an ideal that C++ as a language can only approximate. But it is an ideal that C++ programmers (especially library builders) can strive for. Over the years, the set of language features, standard-library components, and techniques supporting that ideal has grown. Outside of low-level sections of code (hopefully isolated by type-safe interfaces), code that interfaces to code obeying different language conventions (e.g., an operating system call interface), and the implementations of fundamental abstractions (e.g., **string** and **vector**), there is now little need for type-unsafe code.

1.2.3 C Compatibility

C++ was developed from the C programming language and, with few exceptions, retains C as a subset. The main reasons for relying on C were to build on a proven set of low-level language facilities and to be part of a technical community. Great importance was attached to retaining a high degree of compatibility with C [Koenig,1989] [Stroustrup,1994] (Chapter 44); this (unfortunately) precluded cleaning up the C syntax. The continuing, more or less parallel evolution of C and C++ has been a constant source of concern and requires constant attention [Stroustrup,2002]. Having two committees devoted to keeping two widely used languages "as compatible as possible" is not a particularly good way of organizing work. In particular, there are differences in opinion as to the value of compatibility, differences in opinion on what constitutes good programming, and differences in opinion on what support is needed for good programming. Just keeping up communication between the committees is a large amount of work.

One hundred percent C/C++ compatibility was never a goal for C++ because that would compromise type safety and the smooth integration of user-defined and built-in types. However, the definition of C++ has been repeatedly reviewed to remove gratuitous incompatibilities; C++ is now more compatible with C than it was originally. C++98 adopted many details from C89 (§44.3.1). When C then evolved from C89 [C,1990] to C99 [C,1999], C++ adopted almost all of the new features, leaving out VLAs (variable-length arrays) as a misfeature and designated initializers as redundant. C's facilities for low-level systems programming tasks are retained and enhanced; for example, see inlining (§3.2.1.1, §12.1.5, §16.2.8) and **constexpr** (§2.2.3, §10.4, §12.1.6).

Conversely, modern C has adopted (with varying degrees of faithfulness and effectiveness) many features from C++ (e.g., **const**, function prototypes, and inlining; see [Stroustrup,2002]).

The definition of C++ has been revised to ensure that a construct that is both legal C and legal C++ has the same meaning in both languages (§44.3).

One of the original aims for C was to replace assembly coding for the most demanding systems programming tasks. When C++ was designed, care was taken not to compromise the gains in this

area. The difference between C and C++ is primarily in the degree of emphasis on types and structure. C is expressive and permissive. Through extensive use of the type system, C++ is even more expressive without loss of performance.

Knowing C is not a prerequisite for learning C++. Programming in C encourages many techniques and tricks that are rendered unnecessary by C++ language features. For example, explicit type conversion (casting) is less frequently needed in C++ than it is in C (§1.3.3). However, *good* C programs tend to be C++ programs. For example, every program in Kernighan and Ritchie, *The C Programming Language, Second Edition* [Kernighan,1988], is a C++ program. Experience with any statically typed language will be a help when learning C++.

1.2.4 Language, Libraries, and Systems

The C++ fundamental (built-in) types, operators, and statements are those that computer hardware deals with directly: numbers, characters, and addresses. C++ has no built-in high-level data types and no high-level primitive operations. For example, the C++ language does not provide a matrix type with an inversion operator or a string type with a concatenation operator. If a user wants such a type, it can be defined in the language itself. In fact, defining a new general-purpose or application-specific type is the most fundamental programming activity in C++. A well-designed user-defined type differs from a built-in type only in the way it is defined, not in the way it is used. The C++ standard library (Chapter 4, Chapter 5, Chapter 30, Chapter 31, etc.) provides many examples of such types and their uses. From a user's point of view, there is little difference between a built-in type and a type provided by the standard library. Except for a few unfortunate and unimportant historical accidents, the C++ standard library is written in C++. Writing the C++ standard library in C++ is a crucial test of the C++ type system and abstraction mechanisms: they must be (and are) sufficiently powerful (expressive) and efficient (affordable) for the most demanding systems programming tasks. This ensures that they can be used in large systems that typically consist of layer upon layer of abstraction.

Features that would incur run-time or memory overhead even when not used were avoided. For example, constructs that would make it necessary to store "housekeeping information" in every object were rejected, so if a user declares a structure consisting of two 16-bit quantities, that structure will fit into a 32-bit register. Except for the **new**, **delete**, **typeid**, **dynamic_cast**, and **throw** operators, and the **try**-block, individual C++ expressions and statements need no run-time support. This can be essential for embedded and high-performance applications. In particular, this implies that the C++ abstraction mechanisms are usable for embedded, high-performance, high-reliability, and real-time applications. So, programmers of such applications don't have to work with a low-level (error-prone, impoverished, and unproductive) set of language features.

C++ was designed to be used in a traditional compilation and run-time environment: the C programming environment on the UNIX system [UNIX,1985]. Fortunately, C++ was never restricted to UNIX; it simply used UNIX and C as a model for the relationships among language, libraries, compilers, linkers, execution environments, etc. That minimal model helped C++ to be successful on essentially every computing platform. There are, however, good reasons for using C++ in environments that provide significantly more run-time support. Facilities such as dynamic loading, incremental compilation, and a database of type definitions can be put to good use without affecting the language.

Not every piece of code can be well structured, hardware-independent, easy to read, etc. C++ possesses features that are intended for manipulating hardware facilities in a direct and efficient way without concerns for safety or ease of comprehension. It also possesses facilities for hiding such code behind elegant and safe interfaces.

Naturally, the use of C++ for larger programs leads to the use of C++ by groups of programmers. C++'s emphasis on modularity, strongly typed interfaces, and flexibility pays off here. However, as programs get larger, the problems associated with their development and maintenance shift from being language problems to being more global problems of tools and management.

This book emphasizes techniques for providing general-purpose facilities, generally useful types, libraries, etc. These techniques will serve programmers of small programs as well as programmers of large ones. Furthermore, because all nontrivial programs consist of many semi-independent parts, the techniques for writing such parts serve programmers of all applications.

I use the implementation and use of standard-library components, such as **vector**, as examples. This introduces library components and their underlying design concepts and implementation techniques. Such examples show how programmers might design and implement their own libraries. However, if the standard library provides a component that addresses a problem, it is almost always better to use that component than to build your own. Even if the standard component is arguably slightly inferior to a home-built component for a particular problem, the standard component is likely to be more widely applicable, more widely available, and more widely known. Over the longer term, the standard component (possibly accessed through a convenient custom interface) is likely to lower long-term maintenance, porting, tuning, and education costs.

You might suspect that specifying a program by using a more detailed type structure would increase the size of the program source text (or even the size of the generated code). With C++, this is not so. A C++ program declaring function argument types, using classes, etc., is typically a bit shorter than the equivalent C program not using these facilities. Where libraries are used, a C++ program will appear much shorter than its C equivalent, assuming, of course, that a functioning C equivalent could have been built.

C++ supports systems programming. This implies that C++ code is able to effectively interoperate with software written in other languages on a system. The idea of writing all software in a single language is a fantasy. From the beginning, C++ was designed to interoperate simply and efficiently with C, assembler, and Fortran. By that, I meant that a C++, C, assembler, or Fortran function could call functions in the other languages without extra overhead or conversion of data structures passed among them.

C++ was designed to operate within a single address space. The use of multiple processes and multiple address spaces relied on (extralinguistic) operating system support. In particular, I assumed that a C++ programmer would have the operating systems command language available for composing processes into a system. Initially, I relied on the UNIX Shell for that, but just about any "scripting language" will do. Thus, C++ provided no support for multiple address spaces and no support for multiple processes, but it was used for systems relying on those features from the earliest days. C++ was designed to be part of large, concurrent, multilanguage systems.

1.3 Learning C++

No programming language is perfect. Fortunately, a programming language does not have to be perfect to be a good tool for building great systems. In fact, a general-purpose programming language cannot be perfect for all of the many tasks to which it is put. What is perfect for one task is often seriously flawed for another because perfection in one area implies specialization. Thus, C++ was designed to be a good tool for building a wide variety of systems and to allow a wide variety of ideas to be expressed directly.

Not everything can be expressed directly using the built-in features of a language. In fact, that isn't even the ideal. Language features exist to support a variety of programming styles and techniques. Consequently, the task of learning a language should focus on mastering the native and natural styles for that language – not on understanding of every little detail of every language feature. Writing programs is essential; understanding a programming language is not just an intellectual exercise. Practical application of ideas is necessary.

In practical programming, there is little advantage in knowing the most obscure language features or using the largest number of features. A single language feature in isolation is of little interest. Only in the context provided by techniques and by other features does the feature acquire meaning and interest. Thus, when reading the following chapters, please remember that the real purpose of examining the details of C++ is to be able to use language features and library facilities in concert to support good programming styles in the context of sound designs.

No significant system is built exclusively in terms of the language features themselves. We build and use libraries to simplify the task of programming and to increase the quality of our systems. We use libraries to improve maintainability, portability, and performance. Fundamental application concepts are represented as abstractions (e.g., classes, templates, and class hierarchies) in libraries. Many of the most fundamental programming concepts are represented in the standard library. Thus, learning the standard library is an integral part of learning C++. The standard library is the repository of much hard-earned knowledge of how to use C++ well.

C++ is widely used for teaching and research. This has surprised some who – correctly – point out that C++ isn't the smallest or cleanest language ever designed. It is, however:

- Sufficiently clean for successfully teaching basic design and programming concepts
- Sufficiently comprehensive to be a vehicle for teaching advanced concepts and techniques
- Sufficiently realistic, efficient, and flexible for demanding projects
- Sufficiently commercial to be a vehicle for putting what is learned into nonacademic use
- Sufficiently available for organizations and collaborations relying on diverse development and execution environments

C++ is a language that you can grow with.

The most important thing to do when learning C++ is to focus on fundamental concepts (such as type safety, resource management, and invariants) and programming techniques (such as resource management using scoped objects and the use of iterators in algorithms) and not get lost in language-technical details. The purpose of learning a programming language is to become a better programmer, that is, to become more effective at designing and implementing new systems and at maintaining old ones. For this, an appreciation of programming and design techniques is far more important than understanding all the details. The understanding of technical details comes with time and practice.

C++ programming is based on strong static type checking, and most techniques aim at achieving a high level of abstraction and a direct representation of the programmer's ideas. This can usually be done without compromising run-time and space efficiency compared to lower-level techniques. To gain the benefits of C++, programmers coming to it from a different language must learn and internalize idiomatic C++ programming style and technique. The same applies to programmers used to earlier and less expressive versions of C++.

Thoughtlessly applying techniques effective in one language to another typically leads to awkward, poorly performing, and hard-to-maintain code. Such code is also most frustrating to write because every line of code and every compiler error message reminds the programmer that the language used differs from "the old language." You can write in the style of Fortran, C, Lisp, Java, etc., in any language, but doing so is neither pleasant nor economical in a language with a different philosophy. Every language can be a fertile source of ideas about how to write C++ programs. However, ideas must be transformed into something that fits with the general structure and type system of C++ in order to be effective in C++. Over the basic type system of a language, only Pyrrhic victories are possible.

In the continuing debate on whether one needs to learn C before C++, I am firmly convinced that it is best to go directly to C++. C++ is safer and more expressive, and it reduces the need to focus on low-level techniques. It is easier for you to learn the trickier parts of C that are needed to compensate for its lack of higher-level facilities after you have been exposed to the common subset of C and C++ and to some of the higher-level techniques supported directly in C++. Chapter 44 is a guide for programmers going from C++ to C, say, to deal with legacy code. My opinion on how to teach C++ to novices is represented by [Stroustrup,2008].

There are several independently developed implementations of C++. They are supported by a wealth of tools, libraries, and software development environments. To help master all of this you can find textbooks, manuals, and a bewildering variety of online resources. If you plan to use C++ seriously, I strongly suggest that you obtain access to several such sources. Each has its own emphasis and bias, so use at least two.

1.3.1 Programming in C++

The question "How does one write good programs in C++?" is very similar to the question "How does one write good English prose?" There are two answers: "Know what you want to say" and "Practice. Imitate good writing." Both appear to be as appropriate for C++ as they are for English – and as hard to follow.

The main ideal for C++ programming – as for programming in most higher-level languages – is to express concepts (ideas, notions, etc.) from a design directly in code. We try to ensure that the concepts we talk about, represent with boxes and arrows on our whiteboard, and find in our (non-programming) textbooks have direct and obvious counterparts in our programs:

[1] Represent ideas directly in code.
[2] Represent relationships among ideas directly in code (e.g., hierarchical, parametric, and ownership relationships).
[3] Represent independent ideas independently in code.
[4] Keep simple things simple (without making complex things impossible).

More specifically:

[5] Prefer statically type-checked solutions (when applicable).
[6] Keep information local (e.g., avoid global variables, minimize the use of pointers).
[7] Don't overabstract (i.e., don't generalize, introduce class hierarchies, or parameterize beyond obvious needs and experience).

More specific suggestions are listed in §1.3.2.

1.3.2 Suggestions for C++ Programmers

By now, many people have been using C++ for a decade or two. Many more are using C++ in a single environment and have learned to live with the restrictions imposed by early compilers and first-generation libraries. Often, what an experienced C++ programmer has failed to notice over the years is not the introduction of new features as such, but rather the changes in relationships between features that make fundamental new programming techniques feasible. In other words, what you didn't think of when first learning C++ or found impractical just might be a superior approach today. You find out only by reexamining the basics.

Read through the chapters in order. If you already know the contents of a chapter, you can be done in minutes. If you don't already know the contents, you'll have learned something unexpected. I learned a fair bit writing this book, and I suspect that hardly any C++ programmer knows every feature and technique presented. Furthermore, to use the language well, you need a perspective that brings order to the set of features and techniques. Through its organization and examples, this book offers such a perspective.

Take the opportunity offered by the new C++11 facilities to modernize your design and programming techniques:

[1] Use constructors to establish invariants (§2.4.3.2, §13.4, §17.2.1).
[2] Use constructor/destructor pairs to simplify resource management (RAII; §5.2, §13.3).
[3] Avoid "naked" **new** and **delete** (§3.2.1.2, §11.2.1).
[4] Use containers and algorithms rather than built-in arrays and ad hoc code (§4.4, §4.5, §7.4, Chapter 32).
[5] Prefer standard-library facilities to locally developed code (§1.2.4).
[6] Use exceptions, rather than error codes, to report errors that cannot be handled locally (§2.4.3, §13.1).
[7] Use move semantics to avoid copying large objects (§3.3.2, §17.5.2).
[8] Use **unique_ptr** to reference objects of polymorphic type (§5.2.1).
[9] Use **shared_ptr** to reference shared objects, that is, objects without a single owner that is responsible for their destruction (§5.2.1).
[10] Use templates to maintain static type safety (eliminate casts) and avoid unnecessary use of class hierarchies (§27.2).

It might also be a good idea to review the advice for C and Java programmers (§1.3.3, §1.3.4).

1.3.3 Suggestions for C Programmers

The better one knows C, the harder it seems to be to avoid writing C++ in C style, thereby losing many of the potential benefits of C++. Please take a look at Chapter 44, which describes the differences between C and C++.

[1] Don't think of C++ as C with a few features added. C++ can be used that way, but only suboptimally. To get really major advantages from C++ as compared to C, you need to apply different design and implementation styles.
[2] Don't write C in C++; that is often seriously suboptimal for both maintenance and performance.
[3] Use the C++ standard library as a teacher of new techniques and programming styles. Note the difference from the C standard library (e.g., **=** rather than **strcpy()** for copying and **==** rather than **strcmp()** for comparing).
[4] Macro substitution is almost never necessary in C++. Use **const** (§7.5), **constexpr** (§2.2.3, §10.4), **enum** or **enum class** (§8.4) to define manifest constants, **inline** (§12.1.5) to avoid function-calling overhead, **template**s (§3.4, Chapter 23) to specify families of functions and types, and **namespace**s (§2.4.2, §14.3.1) to avoid name clashes.
[5] Don't declare a variable before you need it, and initialize it immediately. A declaration can occur anywhere a statement can (§9.3), in **for**-statement initializers (§9.5), and in conditions (§9.4.3).
[6] Don't use **malloc()**. The **new** operator (§11.2) does the same job better, and instead of **realloc()**, try a **vector** (§3.4.2). Don't just replace **malloc()** and **free()** with "naked" **new** and **delete** (§3.2.1.2, §11.2.1).
[7] Avoid **void***, unions, and casts, except deep within the implementation of some function or class. Their use limits the support you can get from the type system and can harm performance. In most cases, a cast is an indication of a design error. If you must use an explicit type conversion, try using one of the named casts (e.g., **static_cast**; §11.5.2) for a more precise statement of what you are trying to do.
[8] Minimize the use of arrays and C-style strings. C++ standard-library **string**s (§4.2), **array**s (§8.2.4), and **vector**s (§4.4.1) can often be used to write simpler and more maintainable code compared to the traditional C style. In general, try not to build yourself what has already been provided by the standard library.
[9] Avoid pointer arithmetic except in very specialized code (such as a memory manager) and for simple array traversal (e.g., **++p**).
[10] Do not assume that something laboriously written in C style (avoiding C++ features such as classes, templates, and exceptions) is more efficient than a shorter alternative (e.g., using standard-library facilities). Often (but of course not always), the opposite is true.

To obey C linkage conventions, a C++ function must be declared to have C linkage (§15.2.5).

1.3.4 Suggestions for Java Programmers

C++ and Java are rather different languages with similar syntaxes. Their aims are significantly different and so are many of their application domains. Java is *not* a direct successor to C++ in the sense of a language that can do the same as its predecessor, but better and also more. To use C++ well, you need to adopt programming and design techniques appropriate to C++, rather than trying to write Java in C++. It is not just an issue of remembering to **delete** objects that you create with **new** because you can't rely on the presence of a garbage collector:

[1] Don't simply mimic Java style in C++; that is often seriously suboptimal for both maintainability and performance.

[2] Use the C++ abstraction mechanisms (e.g., classes and templates): don't fall back to a C style of programming out of a false feeling of familiarity.
[3] Use the C++ standard library as a teacher of new techniques and programming styles.
[4] Don't immediately invent a unique base for all of your classes (an **Object** class). Typically, you can do better without it for many/most classes.
[5] Minimize the use of reference and pointer variables: use local and member variables (§3.2.1.2, §5.2, §16.3.4, §17.1).
[6] Remember: a variable is never implicitly a reference.
[7] Think of pointers as C++'s equivalent to Java references (C++ references are more limited; there is no reseating of C++ references).
[8] A function is not **virtual** by default. Not every class is meant for inheritance.
[9] Use abstract classes as interfaces to class hierarchies; avoid "brittle base classes," that is, base classes with data members.
[10] Use scoped resource management ("Resource Acquisition Is Initialization"; RAII) whenever possible.
[11] Use a constructor to establish a class invariant (and throw an exception if it can't).
[12] If a cleanup action is needed when an object is deleted (e.g., goes out of scope), use a destructor for that. Don't imitate **finally** (doing so is more ad hoc and in the longer run far more work than relying on destructors).
[13] Avoid "naked" **new** and **delete**; instead, use containers (e.g., **vector**, **string**, and **map**) and handle classes (e.g., **lock** and **unique_ptr**).
[14] Use freestanding functions (nonmember functions) to minimize coupling (e.g., see the standard algorithms), and use namespaces (§2.4.2, Chapter 14) to limit the scope of freestanding functions.
[15] Don't use exception specifications (except **noexcept**; §13.5.1.1).
[16] A C++ nested class does not have access to an object of the enclosing class.
[17] C++ offers only the most minimal run-time reflection: **dynamic_cast** and **typeid** (Chapter 22). Rely more on compile-time facilities (e.g., compile-time polymorphism; Chapter 27, Chapter 28).

Most of this advice applies equally to C# programmers.

1.4 History

I invented C++, wrote its early definitions, and produced its first implementation. I chose and formulated the design criteria for C++, designed its major language features, developed or helped to develop many of the early libraries, and was responsible for the processing of extension proposals in the C++ standards committee.

C++ was designed to provide Simula's facilities for program organization [Dahl,1970] [Dahl,1972] together with C's efficiency and flexibility for systems programming [Kernighan,1978] [Kernighan,1988]. Simula is the initial source of C++'s abstraction mechanisms. The class concept (with derived classes and virtual functions) was borrowed from it. However, templates and exceptions came to C++ later with different sources of inspiration.

The evolution of C++ was always in the context of its use. I spent a lot of time listening to users and seeking out the opinions of experienced programmers. In particular, my colleagues at AT&T Bell Laboratories were essential for the growth of C++ during its first decade.

This section is a brief overview; it does not try to mention every language feature and library component. Furthermore, it does not go into details. For more information, and in particular for more names of people who contributed, see [Stroustrup,1993], [Stroustrup,2007], and [Stroustrup,1994]. My two papers from the ACM History of Programming Languages conference and my *Design and Evolution of C++* book (known as "D&E") describe the design and evolution of C++ in detail and document influences from other programming languages.

Most of the documents produced as part of the ISO C++ standards effort are available online [WG21]. In my FAQ, I try to maintain a connection between the standard facilities and the people who proposed and refined those facilities [Stroustrup,2010]. C++ is not the work of a faceless, anonymous committee or of a supposedly omnipotent "dictator for life"; it is the work of many dedicated, experienced, hard-working individuals.

1.4.1 Timeline

The work that led to C++ started in the fall of 1979 under the name "C with Classes." Here is a simplified timeline:

1979 Work on "C with Classes" started. The initial feature set included classes and derived classes, public/private access control, constructors and destructors, and function declarations with argument checking. The first library supported non-preemptive concurrent tasks and random number generators.

1984 "C with Classes" was renamed to C++. By then, C++ had acquired virtual functions, function and operator overloading, references, and the I/O stream and complex number libraries.

1985 First commercial release of C++ (October 14). The library included I/O streams, complex numbers, and tasks (nonpreemptive scheduling).

1985 *The C++ Programming Language* ("TC++PL," October 14) [Stroustrup,1986].

1989 *The Annotated C++ Reference Manual* ("the ARM").

1991 *The C++ Programming Language, Second Edition* [Stroustrup,1991], presenting generic programming using templates and error handling based on exceptions (including the "Resource Acquisition Is Initialization" general resource management idiom).

1997 *The C++ Programming Language, Third Edition* [Stroustrup,1997] introduced ISO C++, including namespaces, **dynamic_cast**, and many refinements of templates. The standard library added the STL framework of generic containers and algorithms.

1998 ISO C++ standard.

2002 Work on a revised standard, colloquially named C++0x, started.

2003 A "bug fix" revision of the ISO C++ standard was issued. A C++ Technical Report introduced new standard-library components, such as regular expressions, unordered containers (hash tables), and resource management pointers, which later became part of C++0x.

2006 An ISO C++ Technical Report on Performance was issued to answer questions of cost, predictability, and techniques, mostly related to embedded systems programming.

2009 C++0x was feature complete. It provided uniform initialization, move semantics, variadic template arguments, lambda expressions, type aliases, a memory model suitable for concurrency, and much more. The standard library added several components, including threads, locks, and most of the components from the 2003 Technical Report.
2011 ISO C++11 standard was formally approved.
2012 The first complete C++11 implementations emerged.
2012 Work on future ISO C++ standards (referred to as C++14 and C++17) started.
2013 *The C++ Programming Language, Fourth Edition* introduced C++11.

During development, C++11 was known as C++0x. As is not uncommon in large projects, we were overly optimistic about the completion date.

1.4.2 The Early Years

I originally designed and implemented the language because I wanted to distribute the services of a UNIX kernel across multiprocessors and local-area networks (what are now known as multicores and clusters). For that, I needed some event-driven simulations for which Simula would have been ideal, except for performance considerations. I also needed to deal directly with hardware and provide high-performance concurrent programming mechanisms for which C would have been ideal, except for its weak support for modularity and type checking. The result of adding Simula-style classes to C, "C with Classes," was used for major projects in which its facilities for writing programs that use minimal time and space were severely tested. It lacked operator overloading, references, virtual functions, templates, exceptions, and many, many details [Stroustrup,1982]. The first use of C++ outside a research organization started in July 1983.

The name C++ (pronounced "see plus plus") was coined by Rick Mascitti in the summer of 1983 and chosen as the replacement for "C with Classes" by me. The name signifies the evolutionary nature of the changes from C; "++" is the C increment operator. The slightly shorter name "C+" is a syntax error; it had also been used as the name of an unrelated language. Connoisseurs of C semantics find C++ inferior to ++C. The language was not called D, because it was an extension of C, because it did not attempt to remedy problems by removing features, and because there already existed several would-be C successors named D. For yet another interpretation of the name C++, see the appendix of [Orwell,1949].

C++ was designed primarily so that my friends and I would not have to program in assembler, C, or various then-fashionable high-level languages. Its main purpose was to make writing good programs easier and more pleasant for the individual programmer. In the early years, there was no C++ paper design; design, documentation, and implementation went on simultaneously. There was no "C++ project" either, or a "C++ design committee." Throughout, C++ evolved to cope with problems encountered by users and as a result of discussions among my friends, my colleagues, and me.

1.4.2.1 Language Features and Library Facilities

The very first design of C++ (then called "C with Classes") included function declarations with argument type checking and implicit conversions, classes with the **public**/**private** distinction between the interface and the implementation, derived classes, and constructors and destructors. I used macros to provide primitive parameterization. This was in use by mid-1980. Late that year, I was

able to present a set of language facilities supporting a coherent set of programming styles; see §1.2.1. In retrospect, I consider the introduction of constructors and destructors most significant. In the terminology of the time, "a constructor creates the execution environment for the member functions and the destructor reverses that." Here is the root of C++'s strategies for resource management (causing a demand for exceptions) and the key to many techniques for making user code short and clear. If there were other languages at the time that supported multiple constructors capable of executing general code, I didn't (and don't) know of them. Destructors were new in C++.

C++ was released commercially in October 1985. By then, I had added inlining (§12.1.5, §16.2.8), **const**s (§2.2.3, §7.5, §16.2.9), function overloading (§12.3), references (§7.7), operator overloading (§3.2.1.1, Chapter 18, Chapter 19), and virtual functions (§3.2.3, §20.3.2). Of these features, support for run-time polymorphism in the form of virtual functions was by far the most controversial. I knew its worth from Simula but found it impossible to convince most people in the systems programming world of its value. Systems programmers tended to view indirect function calls with suspicion, and people acquainted with other languages supporting object-oriented programming had a hard time believing that **virtual** functions could be fast enough to be useful in systems code. Conversely, many programmers with an object-oriented background had (and many still have) a hard time getting used to the idea that you use virtual function calls only to express a choice that must be made at run time. The resistance to virtual functions may be related to a resistance to the idea that you can get better systems through more regular structure of code supported by a programming language. Many C programmers seem convinced that what really matters is complete flexibility and careful individual crafting of every detail of a program. My view was (and is) that we need every bit of help we can get from languages and tools: the inherent complexity of the systems we are trying to build is always at the edge of what we can express.

Much of the design of C++ was done on the blackboards of my colleagues. In the early years, the feedback from Stu Feldman, Alexander Fraser, Steve Johnson, Brian Kernighan, Doug McIlroy, and Dennis Ritchie was invaluable.

In the second half of the 1980s, I continued to add language features in response to user comments. The most important of those were templates [Stroustrup,1988] and exception handling [Koenig,1990], which were considered experimental at the time the standards effort started. In the design of templates, I was forced to decide among flexibility, efficiency, and early type checking. At the time, nobody knew how to simultaneously get all three, and to compete with C-style code for demanding systems applications, I felt that I had to choose the first two properties. In retrospect, I think the choice was the correct one, and the search for better type checking of templates continues [Gregor,2006] [Sutton,2011] [Stroustrup,2012a]. The design of exceptions focused on multilevel propagation of exceptions, the passing of arbitrary information to an error handler, and the integrations between exceptions and resource management by using local objects with destructors to represent and release resources (what I clumsily called "Resource Acquisition Is Initialization"; §13.3).

I generalized C++'s inheritance mechanisms to support multiple base classes [Stroustrup,1987a]. This was called *multiple inheritance* and was considered difficult and controversial. I considered it far less important than templates or exceptions. Multiple inheritance of abstract classes (often called *interfaces*) is now universal in languages supporting static type checking and object-oriented programming.

The C++ language evolved hand in hand with some of the key library facilities presented in this book. For example, I designed the complex [Stroustrup,1984], vector, stack, and (I/O) stream [Stroustrup,1985] classes together with the operator overloading mechanisms. The first string and list classes were developed by Jonathan Shopiro and me as part of the same effort. Jonathan's string and list classes were the first to see extensive use as part of a library. The string class from the standard C++ library has its roots in these early efforts. The task library described in [Stroustrup,1987b] was part of the first "C with Classes" program ever written in 1980. I wrote it and its associated classes to support Simula-style simulations. Unfortunately, we had to wait until 2011 (30 years!) to get concurrency support standardized and universally available (§1.4.4.2, §5.3, Chapter 41). The development of the template facility was influenced by a variety of **vector**, **map**, **list**, and **sort** templates devised by Andrew Koenig, Alex Stepanov, me, and others.

C++ grew up in an environment with a multitude of established and experimental programming languages (e.g., Ada [Ichbiah,1979], Algol 68 [Woodward,1974], and ML [Paulson,1996]). At the time, I was comfortable in about 25 languages, and their influences on C++ are documented in [Stroustrup,1994] and [Stroustrup,2007]. However, the determining influences always came from the applications I encountered. That was a deliberate policy to have the development of C++ "problem driven" rather than imitative.

1.4.3 The 1998 Standard

The explosive growth of C++ use caused some changes. Sometime during 1987, it became clear that formal standardization of C++ was inevitable and that we needed to start preparing the ground for a standardization effort [Stroustrup,1994]. The result was a conscious effort to maintain contact between implementers of C++ compilers and major users. This was done through paper and electronic mail and through face-to-face meetings at C++ conferences and elsewhere.

AT&T Bell Labs made a major contribution to C++ and its wider community by allowing me to share drafts of revised versions of the C++ reference manual with implementers and users. Because many of those people worked for companies that could be seen as competing with AT&T, the significance of this contribution should not be underestimated. A less enlightened company could have caused major problems of language fragmentation simply by doing nothing. As it happened, about a hundred individuals from dozens of organizations read and commented on what became the generally accepted reference manual and the base document for the ANSI C++ standardization effort. Their names can be found in *The Annotated C++ Reference Manual* ("the ARM") [Ellis,1989]. The X3J16 committee of ANSI was convened in December 1989 at the initiative of Hewlett-Packard. In June 1991, this ANSI (American national) standardization of C++ became part of an ISO (international) standardization effort for C++ and named WG21. From 1990, these joint C++ standards committees have been the main forum for the evolution of C++ and the refinement of its definition. I served on these committees throughout. In particular, as the chairman of the working group for extensions (later called the evolution group), I was directly responsible for handling proposals for major changes to C++ and the addition of new language features. An initial draft standard for public review was produced in April 1995. The first ISO C++ standard (ISO/IEC 14882-1998) [C++,1998] was ratified by a 22-0 national vote in 1998. A "bug fix release" of this standard was issued in 2003, so you sometimes hear people refer to C++03, but that is essentially the same language as C++98.

1.4.3.1 Language Features

By the time the ANSI and ISO standards efforts started, most major language features were in place and documented in the ARM [Ellis,1989]. Consequently, most of the work involved refinement of features and their specification. The template mechanisms, in particular, benefited from much detailed work. Namespaces were introduced to cope with the increased size of C++ programs and the increased number of libraries. At the initiative of Dmitry Lenkov from Hewett-Packard, minimal facilities to use run-time type information (RTTI; Chapter 22) were introduced. I had left such facilities out of C++ because I had found them seriously overused in Simula. I tried to get a facility for optional conservative garbage collection accepted, but failed. We had to wait until the 2011 standard for that.

Clearly, the 1998 language was far superior in features and in particular in the detail of specification to the 1989 language. However, not all changes were improvements. In addition to the inevitable minor mistakes, two major features were added that in retrospect should not have been:

- Exception specifications provide run-time enforcement of which exceptions a function is allowed to throw. They were added at the energetic initiative of people from Sun Microsystems. Exception specifications turned out to be worse than useless for improving readability, reliability, and performance. They are deprecated (scheduled for future removal) in the 2011 standard. The 2011 standard introduced **noexcept** (§13.5.1.1) as a simpler solution to many of the problems that exception specifications were supposed to address.
- It was always obvious that separate compilation of templates and their uses would be ideal [Stroustrup,1994]. How to achieve that under the constraints from real-world uses of templates was not at all obvious. After a long debate in the committee, a compromise was reached and something called **export** templates were specified as part of the 1998 standard. It was not an elegant solution to the problem, only one vendor implemented **export** (the Edison Design Group), and the feature was removed from the 2011 standard. We are still looking for a solution. My opinion is that the fundamental problem is not separate compilation in itself, but that the distinction between interface and implementation of a template is not well specified. Thus, **export** solved the wrong problem. In the future, language support for "concepts" (§24.3) may help by providing precise specification of template requirements. This is an area of active research and design [Sutton,2011] [Stroustrup,2012a].

1.4.3.2 The Standard Library

The greatest and most important innovation in the 1998 standard was the inclusion of the STL, a framework of algorithms and containers, in the standard library (§4.4, §4.5, Chapter 31, Chapter 32, Chapter 33). It was the work of Alex Stepanov (with Dave Musser, Meng Le, and others) based on more than a decade's work on generic programming. Andrew Koenig, Beman Dawes, and I did much to help get the STL accepted [Stroustrup,2007]. The STL has been massively influential within the C++ community and beyond.

Except for the STL, the standard library was a bit of a hodgepodge of components, rather than a unified design. I had failed to ship a sufficiently large foundation library with Release 1.0 of C++ [Stroustrup,1993], and an unhelpful (non-research) AT&T manager had prevented my colleagues and me from rectifying that mistake for Release 2.0. That meant that every major organization (such as Borland, IBM, Microsoft, and Texas Instruments) had its own foundation library by the

time the standards work started. Thus, the committee was limited to a patchwork of components based on what had always been available (e.g., the **complex** library), what could be added without interfering with the major vendor's libraries, and what was needed to ensure cooperation among different nonstandard libraries.

The standard-library **string** (§4.2, Chapter 36) had its origins in early work by Jonathan Shopiro and me at Bell Labs but was revised and extended by several different individuals and groups during standardization. The **valarray** library for numerical computation (§40.5) is primarily the work of Kent Budge. Jerry Schwarz transformed my streams library (§1.4.2.1) into the **iostream**s library (§4.3, Chapter 38) using Andrew Koenig's manipulator technique (§38.4.5.2) and other ideas. The **iostream**s library was further refined during standardization, where the bulk of the work was done by Jerry Schwarz, Nathan Myers, and Norihiro Kumagai.

By commercial standards the C++98 standard library is tiny. For example, there is no standard GUI, database access library, or Web application library. Such libraries are widely available but are not part of the ISO standard. The reasons for that are practical and commercial, rather than technical. However, the C standard library was (and is) many influential people's measure of a standard library, and compared to that, the C++ standard library is huge.

1.4.4 The 2011 Standard

The current C++, C++11, known for years as C++0x, is the work of the members of WG21. The committee worked under increasingly onerous self-imposed processes and procedures. These processes probably led to a better (and more rigorous) specification, but they also limited innovation [Stroustrup,2007]. An initial draft standard for public review was produced in 2009. The second ISO C++ standard (ISO/IEC 14882-2011) [C++,2011] was ratified by a 21-0 national vote in August 2011.

One reason for the long gap between the two standards is that most members of the committee (including me) were under the mistaken impression that the ISO rules required a "waiting period" after a standard was issued before starting work on new features. Consequently, serious work on new language features did not start until 2002. Other reasons included the increased size of modern languages and their foundation libraries. In terms of pages of standards text, the language grew by about 30% and the standard library by about 100%. Much of the increase was due to more detailed specification, rather than new functionality. Also, the work on a new C++ standard obviously had to take great care not to compromise older code through incompatible changes. There are billions of lines of C++ code in use that the committee must not break.

The overall aims for the C++11 effort were:

- Make C++ a better language for systems programming and library building.
- Make C++ easier to teach and learn.

The aims are documented and detailed in [Stroustrup,2007].

A major effort was made to make concurrent systems programming type-safe and portable. This involved a memory model (§41.2) and a set of facilities for lock-free programming (§41.3), which is primarily the work of Hans Boehm, Brian McKnight, and others. On top of that, we added the **thread**s library. Pete Becker, Peter Dimov, Howard Hinnant, William Kempf, Anthony Williams, and others did massive amounts of work on that. To provide an example of what can be achieved on top of the basic concurrency facilities, I proposed work on "a way to exchange

information between tasks without explicit use of a lock,'' which became **future**s and **async()** (§5.3.5); Lawrence Crowl and Detlef Vollmann did most of the work on that. Concurrency is an area where a complete and detailed listing of who did what and why would require a very long paper. Here, I can't even try.

1.4.4.1 Language Features

The list of language features and standard-library facilities added to C++98 to get C++11 is presented in §44.2. With the exception of concurrency support, every addition to the language could be deemed ''minor,'' but doing so would miss the point: language features are meant to be used in combination to write better programs. By ''better'' I mean easier to read, easier to write, more elegant, less error-prone, more maintainable, faster-running, consuming fewer resources, etc.

Here are what I consider the most widely useful new ''building bricks'' affecting the style of C++11 code with references to the text and their primary authors:

- Control of defaults: **=delete** and **=default**: §3.3.4, §17.6.1, §17.6.4; Lawrence Crowl and Bjarne Stroustrup.
- Deducing the type of an object from its initializer, **auto**: §2.2.2, §6.3.6.1; Bjarne Stroustrup. I first designed and implemented **auto** in 1983 but had to remove it because of C compatibility problems.
- Generalized constant expression evaluation (including literal types), **constexpr**: §2.2.3, §10.4, §12.1.6; Gabriel Dos Reis and Bjarne Stroustrup [DosReis,2010].
- In-class member initializers: §17.4.4; Michael Spertus and Bill Seymour.
- Inheriting constructors: §20.3.5.1; Bjarne Stroustrup, Michael Wong, and Michel Michaud.
- Lambda expressions, a way of implicitly defining function objects at the point of their use in an expression: §3.4.3, §11.4; Jaakko Jarvi.
- Move semantics, a way of transmitting information without copying: §3.3.2, §17.5.2; Howard Hinnant.
- A way of stating that a function may not throw exceptions **noexcept**: §13.5.1.1; David Abrahams, Rani Sharoni, and Doug Gregor.
- A proper name for the null pointer, §7.2.2; Herb Sutter and Bjarne Stroustrup.
- The range-**for** statement: §2.2.5, §9.5.1; Thorsten Ottosen and Bjarne Stroustrup.
- Override controls: **final** and **override**: §20.3.4. Alisdair Meredith, Chris Uzdavinis, and Ville Voutilainen.
- Type aliases, a mechanism for providing an alias for a type or a template. In particular, a way of defining a template by binding some arguments of another template: §3.4.5, §23.6; Bjarne Stroustrup and Gabriel Dos Reis.
- Typed and scoped enumerations: **enum class**: §8.4.1; David E. Miller, Herb Sutter, and Bjarne Stroustrup.
- Universal and uniform initialization (including arbitrary-length initializer lists and protection against narrowing): §2.2.2, §3.2.1.3, §6.3.5, §17.3.1, §17.3.4; Bjarne Stroustrup and Gabriel Dos Reis.
- Variadic templates, a mechanism for passing an arbitrary number of arguments of arbitrary types to a template: §3.4.4, §28.6; Doug Gregor and Jaakko Jarvi.

Many more people than can be listed here deserve to be mentioned. The technical reports to the committee [WG21] and my C++11 FAQ [Stroustrup,2010a] give many of the names. The minutes of the committee's working groups mention more still. The reason my name appears so often is (I hope) not vanity, but simply that I chose to work on what I consider important. These are features that will be pervasive in good code. Their major role is to flesh out the C++ feature set to better support programming styles (§1.2.1). They are the foundation of the synthesis that is C++11.

Much work went into a proposal that did not make it into the standard. "Concepts" was a facility for specifying and checking requirements for template arguments [Gregor,2006] based on previous research (e.g., [Stroustrup,1994] [Siek,2000] [DosReis,2006]) and extensive work in the committee. It was designed, specified, implemented, and tested, but by a large majority the committee decided that the proposal was not yet ready. Had we been able to refine "concepts," it would have been the most important single feature in C++11 (its only competitor for that title is concurrency support). However, the committee decided against "concepts" on the grounds of complexity, difficulty of use, and compile-time performance [Stroustrup,2010b]. I think we (the committee) did the right thing with "concepts" for C++11, but this feature really was "the one that got away." This is currently a field of active research and design [Sutton,2011] [Stroustrup,2012a].

1.4.4.2 Standard Library

The work on what became the C++11 standard library started with a standards committee technical report ("TR1"). Initially, Matt Austern was the head of the Library Working Group, and later Howard Hinnant took over until we shipped the final draft standard in 2011.

As for language features, I'll only list a few standard-library components with references to the text and the names of the individuals most closely associated with them. For a more detailed list, see §44.2.2. Some components, such as **unordered_map** (hash tables), were ones we simply didn't manage to finish in time for the C++98 standard. Many others, such as **unique_ptr** and **function** were part of a technical report (TR1) based on Boost libraries. Boost is a volunteer organization created to provide useful library components based on the STL [Boost].

- Hashed containers, such as **unordered_map**: §31.4.3; Matt Austern.
- The basic concurrency library components, such as **thread**, **mutex**, and **lock**: §5.3, §42.2; Pete Becker, Peter Dimov, Howard Hinnant, William Kempf, Anthony Williams, and more.
- Launching asynchronous computation and returning results, **future**, **promise**, and **async()**: §5.3.5, §42.4.6; Detlef Vollmann, Lawrence Crowl, Bjarne Stroustrup, and Herb Sutter.
- The garbage collection interface: §34.5; Michael Spertus and Hans Boehm.
- A regular expression library, **regexp**: §5.5, Chapter 37; John Maddock.
- A random number library: §5.6.3, §40.7; Jens Maurer and Walter Brown. It was about time. I shipped the first random number library with "C with Classes" in 1980.

Several utility components were tried out in Boost:

- A pointer for simply and efficiently passing resources, **unique_ptr**: §5.2.1, §34.3.1; Howard E. Hinnant. This was originally called **move_ptr** and is what **auto_ptr** should have been had we known how to do so for C++98.
- A pointer for representing shared ownership, **shared_ptr**: §5.2.1, §34.3.2; Peter Dimov. A successor to the C++98 **counted_ptr** proposal from Greg Colvin.

- The **tuple** library: §5.4.3, §28.5, §34.2.4.2; Jaakko Jarvi and Gary Powell. They credit a long list of contributors, including Doug Gregor, David Abrahams, and Jeremy Siek.
- The general **bind()**: §33.5.1; Peter Dimov. His acknowledgments list a veritable who's who of Boost (including Doug Gregor, John Maddock, Dave Abrahams, and Jaakko Jarvi).
- The **function** type for holding callable objects: §33.5.3; Doug Gregor. He credits William Kempf and others with contributions.

1.4.5 What is C++ used for?

By now (2013), C++ is used just about everywhere: it is in your computer, your phone, your car, probably even in your camera. You don't usually see it. C++ is a systems programming language, and its most pervasive uses are deep in the infrastructure where we, as users, never look.

C++ is used by millions of programmers in essentially every application domain. Billions (thousands of millions) of lines of C++ are currently deployed. This massive use is supported by half a dozen independent implementations, many thousands of libraries, hundreds of textbooks, and dozens of websites. Training and education at a variety of levels are widely available.

Early applications tended to have a strong systems programming flavor. For example, several early operating systems have been written in C++: [Campbell,1987] (academic), [Rozier,1988] (real time), [Berg,1995] (high-throughput I/O). Many current ones (e.g., Windows, Apple's OS, Linux, and most portable-device OSs) have key parts done in C++. Your cellphone and Internet routers are most likely written in C++. I consider uncompromising low-level efficiency essential for C++. This allows us to use C++ to write device drivers and other software that rely on direct manipulation of hardware under real-time constraints. In such code, predictability of performance is at least as important as raw speed. Often, so is the compactness of the resulting system. C++ was designed so that every language feature is usable in code under severe time and space constraints (§1.2.4) [Stroustrup,1994,§4.5].

Some of today's most visible and widely used systems have their critical parts written in C++. Examples are Amadeus (airline ticketing), Amazon (Web commerce), Bloomberg (financial information), Google (Web search), and Facebook (social media). Many other programming languages and technologies depend critically on C++'s performance and reliability in their implementation. Examples include the most widely used Java Virtual Machines (e.g., Oracle's HotSpot), JavaScript interpreters (e.g., Google's V8), browsers (e.g., Microsoft's Internet Explorer, Mozilla's Firefox, Apple's Safari, and Google's Chrome), and application frameworks (e.g., Microsoft's .NET Web services framework). I consider C++ to have unique strengths in the area of infrastructure software [Stroustrup,2012a].

Most applications have sections of code that are critical for acceptable performance. However, the largest amount of code is not in such sections. For most code, maintainability, ease of extension, and ease of testing are key. C++'s support for these concerns has led to its widespread use in areas where reliability is a must and where requirements change significantly over time. Examples are financial systems, telecommunications, device control, and military applications. For decades, the central control of the U.S. long-distance telephone system has relied on C++, and every 800 call (i.e., a call paid for by the called party) has been routed by a C++ program [Kamath,1993]. Many such applications are large and long-lived. As a result, stability, compatibility, and scalability have been constant concerns in the development of C++. Multimillion-line C++ programs are common.

Games is another area where a multiplicity of languages and tools need to coexist with a language providing uncompromising efficiency (often on ‘‘unusual’’ hardware). Thus, games has been another major applications area for C++.

What used to be called systems programming is widely found in embedded systems, so it is not surprising to find massive use of C++ in demanding embedded systems projects, including computer tomography (CAT scanners), flight control software (e.g., Lockheed-Martin), rocket control, ship’s engines (e.g., the control of the world’s largest marine diesel engines from MAN), automobile software (e.g., BMW), and wind turbine control (e.g., Vesta).

C++ wasn’t specifically designed with numerical computation in mind. However, much numerical, scientific, and engineering computation is done in C++. A major reason for this is that traditional numerical work must often be combined with graphics and with computations relying on data structures that don’t fit into the traditional Fortran mold (e.g., [Root,1995]). I am particularly pleased to see C++ used in major scientific endeavors, such as the Human Genome Project, NASA’s Mars Rovers, CERN’s search for the fundamentals of the universe, and many others.

C++’s ability to be used effectively for applications that require work in a variety of application areas is an important strength. Applications that involve local- and wide-area networking, numerics, graphics, user interaction, and database access are common. Traditionally, such application areas were considered distinct and were served by distinct technical communities using a variety of programming languages. However, C++ is widely used in all of those areas, and more. It is designed so that C++ code can coexist with code written in other languages. Here, again, C++’s stability over decades is important. Furthermore, no really major system is written 100% in a single language. Thus, C++’s original design aim of interoperability becomes significant.

Major applications are not written in just the raw language. C++ is supported by a variety of libraries (beyond the ISO C++ standard library) and tool sets, such as Boost [Boost] (portable foundation libraries), POCO (Web development), QT (cross-platform application development), wxWidgets (a cross-platform GUI library), WebKit (a layout engine library for Web browsers), CGAL (computational geometry), QuickFix (Financial Information eXchange), OpenCV (real-time image processing), and Root [Root,1995] (High-Energy Physics). There are many thousands of C++ libraries, so keeping up with them all is impossible.

1.5 Advice

Each chapter contains an ‘‘Advice’’ section with a set of concrete recommendations related to its contents. Such advice consists of rough rules of thumb, not immutable laws. A piece of advice should be applied only where reasonable. There is no substitute for intelligence, experience, common sense, and good taste.

I find rules of the form ‘‘never do this’’ unhelpful. Consequently, most advice is phrased as suggestions for what to do. Negative suggestions tend not to be phrased as absolute prohibitions and I try to suggest alternatives. I know of no major feature of C++ that I have not seen put to good use. The ‘‘Advice’’ sections do not contain explanations. Instead, each piece of advice is accompanied by a reference to an appropriate section of the book.

For starters, here are a few high-level recommendations derived from the sections on design, learning, and history of C++:

[1] Represent ideas (concepts) directly in code, for example, as a function, a class, or an enumeration; §1.2.
[2] Aim for your code to be both elegant and efficient; §1.2.
[3] Don't overabstract; §1.2.
[4] Focus design on the provision of elegant and efficient abstractions, possibly presented as libraries; §1.2.
[5] Represent relationships among ideas directly in code, for example, through parameterization or a class hierarchy; §1.2.1.
[6] Represent independent ideas separately in code, for example, avoid mutual dependencies among classes; §1.2.1.
[7] C++ is not just object-oriented; §1.2.1.
[8] C++ is not just for generic programming; §1.2.1.
[9] Prefer solutions that can be statically checked; §1.2.1.
[10] Make resources explicit (represent them as class objects); §1.2.1, §1.4.2.1.
[11] Express simple ideas simply; §1.2.1.
[12] Use libraries, especially the standard library, rather than trying to build everything from scratch; §1.2.1.
[13] Use a type-rich style of programming; §1.2.2.
[14] Low-level code is not necessarily efficient; don't avoid classes, templates, and standard-library components out of fear of performance problems; §1.2.4, §1.3.3.
[15] If data has an invariant, encapsulate it; §1.3.2.
[16] C++ is not just C with a few extensions; §1.3.3.

In general: To write a good program takes intelligence, taste, and patience. You are not going to get it right the first time. Experiment!

1.6 References

[Austern,2003] Matt Austern et al.: *Untangling the Balancing and Searching of Balanced Binary Search Trees*. Software – Practice & Experience. Vol 33, Issue 13. November 2003.

[Barron,1963] D. W. Barron et al.: *The main features of CPL.* The Computer Journal. 6 (2): 134. (1963). comjnl.oxfordjournals.org/content/6/2/134.full.pdf+html.

[Barton,1994] J. J. Barton and L. R. Nackman: *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*. Addison-Wesley. Reading, Massachusetts. 1994. ISBN 0-201-53393-6.

[Berg,1995] William Berg, Marshall Cline, and Mike Girou: *Lessons Learned from the OS/400 OO Project*. CACM. Vol. 38, No. 10. October 1995.

[Boehm,2008] Hans-J. Boehm and Sarita V. Adve: *Foundations of the C++ concurrency memory model*. ACM PLDI'08.

[Boost] The Boost library collection. www.boost.org.

[Budge,1992] Kent Budge, J. S. Perry, and A. C. Robinson: *High-Performance Scientific Computation Using C++*. Proc. USENIX C++ Conference. Portland, Oregon. August 1992.

[C,1990] X3 Secretariat: *Standard – The C Language*. X3J11/90-013. ISO Standard ISO/IEC 9899-1990. Computer and Business Equipment Manufacturers Association. Washington, DC.

[C,1999] ISO/IEC 9899. *Standard – The C Language*. X3J11/90-013-1999.

[C,2011] ISO/IEC 9899. *Standard – The C Language*. X3J11/90-013-2011.

[C++,1998] ISO/IEC JTC1/SC22/WG21: *International Standard – The C++ Language*. ISO/IEC 14882:1998.

[C++Math,2010] *International Standard – Extensions to the C++ Library to Support Mathematical Special Functions*. ISO/IEC 29124:2010.

[C++,2011] ISO/IEC JTC1/SC22/WG21: *International Standard – The C++ Language*. ISO/IEC 14882:2011.

[Campbell,1987] Roy Campbell et al.: *The Design of a Multiprocessor Operating System*. Proc. USENIX C++ Conference. Santa Fe, New Mexico. November 1987.

[Coplien,1995] James O. Coplien: *Curiously Recurring Template Patterns*. The C++ Report. February 1995.

[Cox,2007] Russ Cox: *Regular Expression Matching Can Be Simple And Fast*. January 2007. swtch.com/˜rsc/regexp/regexp1.html.

[Czarnecki,2000] K. Czarnecki and U. Eisenecker: *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley. Reading, Massachusetts. 2000. ISBN 0-201-30977-7.

[Dahl,1970] O-J. Dahl, B. Myrhaug, and K. Nygaard: *SIMULA Common Base Language.* Norwegian Computing Center S-22. Oslo, Norway. 1970.

[Dahl,1972] O-J. Dahl and C. A. R. Hoare: *Hierarchical Program Construction* in *Structured Programming.* Academic Press. New York. 1972.

[Dean,2004] J. Dean and S. Ghemawat: *MapReduce: Simplified Data Processing on Large Clusters*. OSDI'04: Sixth Symposium on Operating System Design and Implementation. 2004.

[Dechev,2010] D. Dechev, P. Pirkelbauer, and B. Stroustrup: *Understanding and Effectively Preventing the ABA Problem in Descriptor-based Lock-free Designs*. 13th IEEE Computer Society ISORC 2010 Symposium. May 2010.

[DosReis,2006] Gabriel Dos Reis and Bjarne Stroustrup: *Specifying C++ Concepts*. POPL06. January 2006.

[DosReis,2010] Gabriel Dos Reis and Bjarne Stroustrup: *General Constant Expressions for System Programming Languages*. SAC-2010. The 25th ACM Symposium On Applied Computing. March 2010.

[DosReis,2011] Gabriel Dos Reis and Bjarne Stroustrup: *A Principled, Complete, and Efficient Representation of C++*. Journal of Mathematics in Computer Science. Vol. 5, Issue 3. 2011.

[Ellis,1989] Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley. Reading, Mass. 1990. ISBN 0-201-51459-1.

[Freeman,1992] Len Freeman and Chris Phillips: *Parallel Numerical Algorithms*. Prentice Hall. Englewood Cliffs, New Jersey. 1992. ISBN 0-13-651597-5.

[Friedl,1997]: Jeffrey E. F. Friedl: *Mastering Regular Expressions*. O'Reilly Media. Sebastopol, California. 1997. ISBN 978-1565922570.

[Gamma,1995] Erich Gamma et al.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Reading, Massachusetts. 1994. ISBN 0-201-63361-2.

[Gregor,2006] Douglas Gregor et al.: *Concepts: Linguistic Support for Generic Programming in C++*. OOPSLA'06.

[Hennessy,2011] John L. Hennessy and David A. Patterson: *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann. San Francisco, California. 2011. ISBN 978-0123838728.

[Ichbiah,1979] Jean D. Ichbiah et al.: *Rationale for the Design of the ADA Programming Language*. SIGPLAN Notices. Vol. 14, No. 6. June 1979.

[Kamath,1993] Yogeesh H. Kamath, Ruth E. Smilan, and Jean G. Smith: *Reaping Benefits with Object-Oriented Technology*. AT&T Technical Journal. Vol. 72, No. 5. September/October 1993.

[Kernighan,1978] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language.* Prentice Hall. Englewood Cliffs, New Jersey. 1978.

[Kernighan,1988] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language, Second Edition.* Prentice-Hall. Englewood Cliffs, New Jersey. 1988. ISBN 0-13-110362-8.

[Knuth,1968] Donald E. Knuth: *The Art of Computer Programming*. Addison-Wesley. Reading, Massachusetts. 1968.

[Koenig,1989] Andrew Koenig and Bjarne Stroustrup: *C++: As close to C as possible – but no closer*. The C++ Report. Vol. 1, No. 7. July 1989.

[Koenig,1990] A. R. Koenig and B. Stroustrup: *Exception Handling for C++ (revised)*. Proc USENIX C++ Conference. April 1990.

[Kolecki,2002] Joseph C. Kolecki: *An Introduction to Tensors for Students of Physics and Engineering*. NASA/TM-2002-211716.

[Langer,2000] Angelika Langer and Klaus Kreft: *Standard C++ IOStreams and Locales: Advanced Programmer's Guide and Reference*. Addison-Wesley. 2000. ISBN 978-0201183955.

[McKenney] Paul E. McKenney: *Is Parallel Programming Hard, And, If So, What Can You Do About It?* kernel.org. Corvallis, Oregon. 2012. http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html.

[Maddock,2009] John Maddock: *Boost.Regex*. www.boost.org. 2009.

[Orwell,1949] George Orwell: *1984.* Secker and Warburg. London. 1949.

[Paulson,1996] Larry C. Paulson: *ML for the Working Programmer*. Cambridge University Press. Cambridge. 1996. ISBN 0-521-56543-X.

[Pirkelbauer,2009] P. Pirkelbauer, Y. Solodkyy, and B. Stroustrup: *Design and Evaluation of C++ Open Multi-Methods*. Science of Computer Programming. Elsevier Journal. June 2009. doi:10.1016/j.scico.2009.06.002.

[Richards,1980] Martin Richards and Colin Whitby-Strevens: *BCPL – The Language and Its Compiler.* Cambridge University Press. Cambridge. 1980. ISBN 0-521-21965-5.

[Root,1995] *ROOT: A Data Analysis Framework*. root.cern.ch. It seems appropriate to represent a tool from CERN, the birthplace of the World Wide Web, by a

Web address.

[Rozier,1988] M. Rozier et al.: *CHORUS Distributed Operating Systems*. Computing Systems. Vol. 1, No. 4. Fall 1988.

[Siek,2000] Jeremy G. Siek and Andrew Lumsdaine: *Concept checking: Binding parametric polymorphism in C++*. Proc. First Workshop on C++ Template Programming. Erfurt, Germany. 2000.

[Solodkyy,2012] Y. Solodkyy, G. Dos Reis, and B. Stroustrup: *Open and Efficient Type Switch for C++*. Proc. OOPSLA'12.

[Stepanov,1994] Alexander Stepanov and Meng Lee: *The Standard Template Library*. HP Labs Technical Report HPL-94-34 (R. 1). 1994.

[Stewart,1998] G. W. Stewart: *Matrix Algorithms, Volume I. Basic Decompositions*. SIAM. Philadelphia, Pennsylvania. 1998.

[Stroustrup,1982] B. Stroustrup: *Classes: An Abstract Data Type Facility for the C Language*. Sigplan Notices. January 1982. The first public description of "C with Classes."

[Stroustrup,1984] B. Stroustrup: *Operator Overloading in C++*. Proc. IFIP WG2.4 Conference on System Implementation Languages: Experience & Assessment. September 1984.

[Stroustrup,1985] B. Stroustrup: *An Extensible I/O Facility for C++*. Proc. Summer 1985 USENIX Conference.

[Stroustrup,1986] B. Stroustrup: *The C++ Programming Language*. Addison-Wesley. Reading, Massachusetts. 1986. ISBN 0-201-12078-X.

[Stroustrup,1987] B. Stroustrup: *Multiple Inheritance for C++*. Proc. EUUG Spring Conference. May 1987.

[Stroustrup,1987b] B. Stroustrup and J. Shopiro: *A Set of C Classes for Co-Routine Style Programming*. Proc. USENIX C++ Conference. Santa Fe, New Mexico. November 1987.

[Stroustrup,1988] B. Stroustrup: *Parameterized Types for C++*. Proc. USENIX C++ Conference, Denver. 1988.

[Stroustrup,1991] B. Stroustrup: *The C++ Programming Language (Second Edition)*. Addison-Wesley. Reading, Massachusetts. 1991. ISBN 0-201-53992-6.

[Stroustrup,1993] B. Stroustrup: *A History of C++: 1979-1991*. Proc. ACM History of Programming Languages conference (HOPL-2). ACM Sigplan Notices. Vol 28, No 3. 1993.

[Stroustrup,1994] B. Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. Reading, Mass. 1994. ISBN 0-201-54330-3.

[Stroustrup,1997] B. Stroustrup: *The C++ Programming Language, Third Edition*. Addison-Wesley. Reading, Massachusetts. 1997. ISBN 0-201-88954-4. Hardcover ("Special") Edition. 2000. ISBN 0-201-70073-5.

[Stroustrup,2002] B. Stroustrup: *C and C++: Siblings*, *C and C++: A Case for Compatibility*, and *C and C++: Case Studies in Compatibility*. The C/C++ Users Journal. July-September 2002. www.stroustrup.com/papers.html.

[Stroustrup,2007] B. Stroustrup: *Evolving a language in and for the real world: C++ 1991-2006*. ACM HOPL-III. June 2007.

[Stroustrup,2008] B. Stroustrup: *Programming – Principles and Practice Using C++*. Addison-Wesley. 2009. ISBN 0-321-54372-6.

[Stroustrup,2010a] B. Stroustrup: *The C++11 FAQ*. www.stroustrup.com/C++11FAQ.html.

[Stroustrup,2010b] B. Stroustrup: *The C++0x "Remove Concepts" Decision*. Dr. Dobb's Journal. July 2009.

[Stroustrup,2012a] B. Stroustrup and A. Sutton: *A Concept Design for the STL*. WG21 Technical Report N3351==12-0041. January 2012.

[Stroustrup,2012b] B. Stroustrup: *Software Development for Infrastructure*. Computer. January 2012. doi:10.1109/MC.2011.353.

[Sutton,2011] A. Sutton and B. Stroustrup: *Design of Concept Libraries for C++*. Proc. SLE 2011 (International Conference on Software Language Engineering). July 2011.

[Tanenbaum,2007] Andrew S. Tanenbaum: *Modern Operating Systems, Third Edition*. Prentice Hall. Upper Saddle River, New Jersey. 2007. ISBN 0-13-600663-9.

[Tsafrir,2009] Dan Tsafrir et al.: *Minimizing Dependencies within Generic Classes for Faster and Smaller Programs*. ACM OOPSLA'09. October 2009.

[Unicode,1996] The Unicode Consortium: *The Unicode Standard, Version 2.0*. Addison-Wesley. Reading, Massachusetts. 1996. ISBN 0-201-48345-9.

[UNIX,1985] *UNIX Time-Sharing System: Programmer's Manual. Research Version, Tenth Edition.* AT&T Bell Laboratories, Murray Hill, New Jersey. February 1985.

[Vandevoorde,2002] David Vandevoorde and Nicolai M. Josuttis: *C++ Templates: The Complete Guide*. Addison-Wesley. 2002. ISBN 0-201-73484-2.

[Veldhuizen,1995] Todd Veldhuizen: *Expression Templates*. The C++ Report. June 1995.

[Veldhuizen,2003] Todd L. Veldhuizen: *C++ Templates are Turing Complete*. Indiana University Computer Science Technical Report. 2003.

[Vitter,1985] Jefferey Scott Vitter: *Random Sampling with a Reservoir*. ACM Transactions on Mathematical Software, Vol. 11, No. 1. 1985.

[WG21] ISO SC22/WG21 The C++ Programming Language Standards Committee: *Document Archive*. www.open-std.org/jtc1/sc22/wg21.

[Williams,2012] Anthony Williams: *C++ Concurrency in Action – Practical Multithreading*. Manning Publications Co. ISBN 978-1933988771.

[Wilson,1996] Gregory V. Wilson and Paul Lu (editors): *Parallel Programming Using C++*. The MIT Press. Cambridge, Mass. 1996. ISBN 0-262-73118-5.

[Wood,1999] Alistair Wood: *Introduction to Numerical Analysis*. Addison-Wesley. Reading, Massachusetts. 1999. ISBN 0-201-34291-X.

[Woodward,1974] P. M. Woodward and S. G. Bond: *Algol 68-R Users Guide.* Her Majesty's Stationery Office. London. 1974.

2

A Tour of C++: The Basics

The first thing we do, let's
kill all the language lawyers.
– Henry VI, Part II

- Introduction
- The Basics
  Hello, World!; Types, Variables, and Arithmetic; Constants; Tests and Loops; Pointers, Arrays, and Loops
- User-Defined Types
  Structures; Classes; Enumerations
- Modularity
  Separate Compilation; Namespaces; Error Handling
- Postscript
- Advice

2.1 Introduction

The aim of this chapter and the next three is to give you an idea of what C++ is, without going into a lot of details. This chapter informally presents the notation of C++, C++'s model of memory and computation, and the basic mechanisms for organizing code into a program. These are the language facilities supporting the styles most often seen in C and sometimes called *procedural programming*. Chapter 3 follows up by presenting C++'s abstraction mechanisms. Chapter 4 and Chapter 5 give examples of standard-library facilities.

The assumption is that you have programmed before. If not, please consider reading a textbook, such as *Programming: Principles and Practice Using C++* [Stroustrup,2009], before continuing here. Even if you have programmed before, the language you used or the applications you wrote may be very different from the style of C++ presented here. If you find this "lightning tour" confusing, skip to the more systematic presentation starting in Chapter 6.