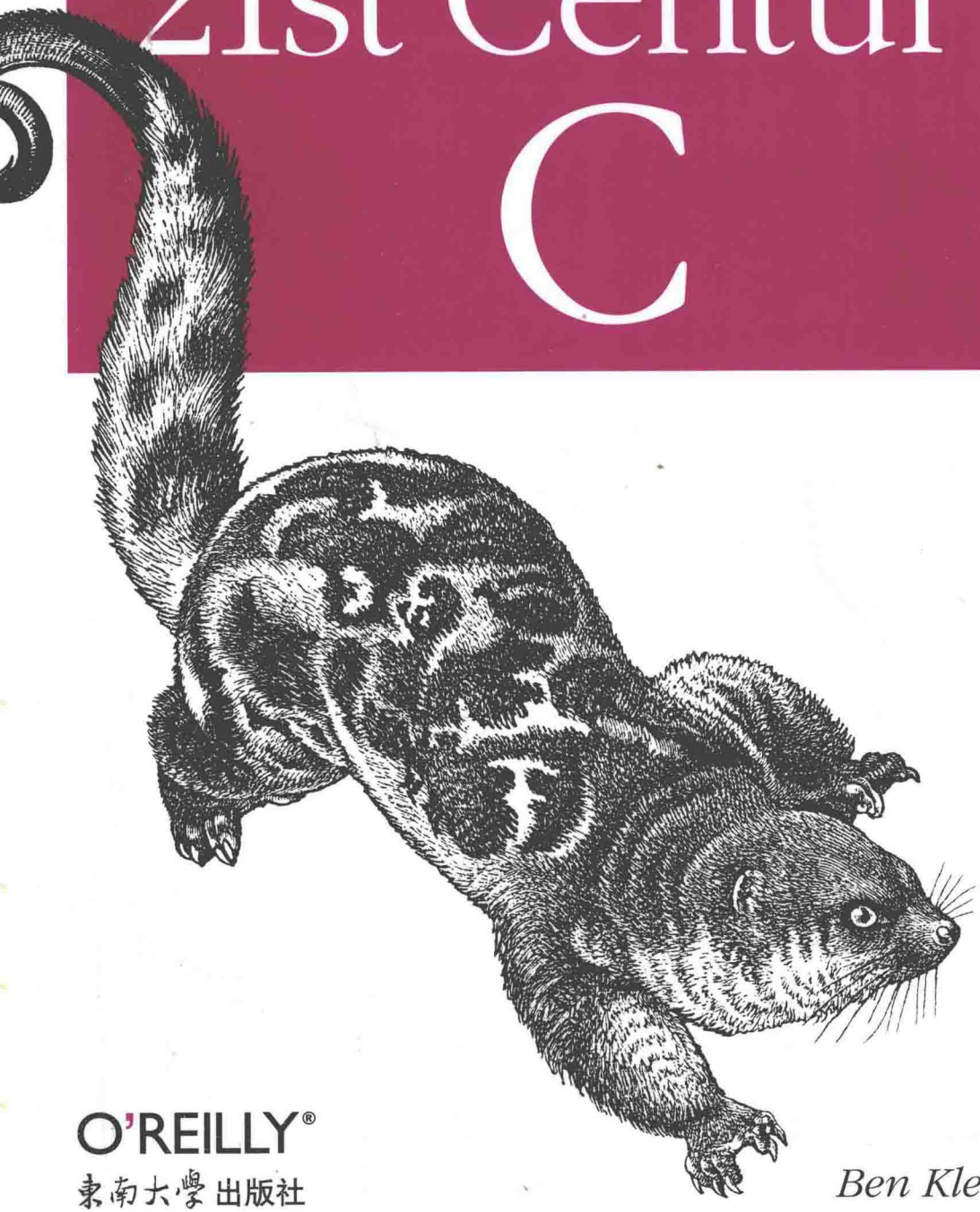
21世纪C语言（影印版）

21st Century C



O'REILLY®
東南大學出版社

Ben Klemens 著

21世纪C语言（影印版）

21st Century C

Ben Klemens 著

O'REILLY®

Beijing · Cambridge · Farnham · Köln · Sebastopol · Tokyo

O'Reilly Media, Inc.授权东南大学出版社出版

南京 东南大学出版社

图书在版编目（CIP）数据

21 世纪 C 语言：英文 /（美)克莱门斯（Klemens, B.）著 . —影印本 . —南京：东南大学出版社，2013.5

书名原文：21st Century C

ISBN 978-7-5641-4205-6

I. ① 2… II. ①克… III. ① C 语言－程序设计－英文 IV. ① TP312

中国版本图书馆 CIP 数据核字（2013）第 097340 号

江苏省版权局著作权合同登记

图字：10-2013-124 号

21 世纪 C 语言（影印版）

出版发行：东南大学出版社
地　　址：南京四牌楼 2 号　　邮编：210096
出 版 人：江建中
网　　址：http://www.seupress.com
电子邮件：press@seupress.com
印　　刷：扬中市印刷有限公司
开　　本：787 毫米 × 980 毫米　16 开本
印　　张：18.5
字　　数：362 千字
版　　次：2013 年 5 月第 1 版
印　　次：2013 年 5 月第 1 次印刷
书　　号：ISBN 978-7-5641-4205-6
定　　价：56.00 元（册）

本社图书若有印装质量问题，请直接与营销部联系。电话（传真）：025-83791830

Preface

Is it really punk rock
Like the party line?

—Wilco, *"Too Far Apart"*

C Is Punk Rock

C has only a handful of keywords and is a bit rough around the edges, and it rocks. You can do anything with it. Like the C, G, and D chords on a guitar, you can learn the basic mechanics pretty quickly, and then spend the rest of your life getting better. The people who don't get it fear its power and think it too edgy to be safe. By all rankings, it is consistently the most popular language that doesn't have a corporation or foundation spending money to promote it.[1]

Also, the language is about 40 years old, which makes it middle-aged. It was written by a few guys basically working against management—the perfect punk rock origins—but that was in the 1970s, and there's been a lot of time for the language to go mainstream.

What did people do when punk rock went mainstream? In the decades since its advent in the 1970s, punk certainly has come in from the fringes: The Clash, The Offspring, Green Day, and The Strokes sold millions of albums worldwide (to name just a few), and I have heard lite instrumental versions of songs from the punk spinoff known as grunge at my local supermarket. The former lead singer of Sleater-Kinney now has a popular sketch comedy show that frequently lampoons punk rockers.[2] One reaction to the continuing evolution would be to take the hard line and say that the original stuff was punk and everything else is just easy punk pop for the masses. The traditionalists can still play their albums from the '70s, and if the grooves are worn out, they can

1. This preface owes an obvious and huge debt to "Punk Rock Languages: A Polemic," by Chris Adamson, at *http://pragprog.com/magazines/2011-03/punk-rock-languages*.

2. With lyrics like "Can't get to heaven with a three-chord song," maybe Sleater-Kinney was post-punk? Unfortunately, there is no ISO Punk standard we can look to for precise in-or-out definitions.

download a digitally mastered edition. They can buy Ramones hoodies for their toddlers.

Outsiders don't get it. Some of them hear the word *punk* and picture something out of the 1970s—a historic artifact about some kids that were, at the time, really doing something different. The traditionalist punks who still love and play their 1973 Iggy Pop LPs are having their fun, but they bolster the impression that punk is ossified and no longer relevant.

Getting back to the world of C, we have both the traditionalists, waving the banner of ANSI '89, and those who will rock out to whatever works and may not even realize that the code they are writing would not have compiled or run in the 1990s. Outsiders don't get the difference. They see still-in-print books from the 1980s and still-online tutorials from the 1990s, they hear from the hardcore traditionalists who insist on still writing like that today, and they don't even know that the language and the rest of its users continue to evolve. That's a shame, because they're missing out on some great stuff.

This is a book about breaking tradition and keeping C punk rock. I really don't care to compare the code in this book to the original C specification in Kernighan & Ritchie's 1978 book. My *telephone* has 512 megabytes of memory, so why are our C textbooks still spending pages upon pages covering techniques to shave kilobytes off of our executables? I am writing this on a bottom-of-the-line red netbook that can accommodate 3,200,000,000 instructions per second; what do I care about whether an operation requires comparing 8 bits or 16? We should be writing code that we can write quickly and that is readable by our fellow humans. We're still writing in C, so our readable but imperfectly optimized code will still run an order of magnitude faster than if we'd written comparable code in any number of alternative, bloated languages.

Q & A (Or, the Parameters of the Book)

Q: How is this C book different from all others?

A: C textbooks are a pretty uniform bunch (I've read *a lot* of them, including [Griffiths 2012], [Kernighan 1978], [Kernighan 1988], [Kochan 2004], [Oualline 1997], [Perry 1994], [Prata 2004], and [Ullman 2004]). Most were written before the C99 standard simplified many aspects of usage, and you can tell that some of those now in their *N*th edition just pasted in a few notes about updates rather than really rethinking how to use the language. They all mention that there might be libraries that you could maybe use in writing your own code, but they predate the installation tools and ecosystem we have now that make using those libraries reliable and reasonably portable. Those textbooks are still valid and still have value, but modern C code just doesn't look like the code in those textbooks.

This book picks up where they left off, reconsidering the language and the ecosystem in which it lives. The storyline here is about using libraries that provide linked lists and

XML parsers, not writing new ones from scratch. It is about writing code that is readable and function interfaces that are user-friendly.

Q: Who is this book for? Do I need to be a coding guru?

A: You have experience coding in any language, maybe Java or a scripting language such as Perl. I don't have to sell you on why your code shouldn't just be one long routine with no subfunctions.

You have some knowledge of C, but don't worry if you don't know it too well—as I'll detail, there's a lot you're better off never learning. If you are a blank slate with respect to C syntax, it really is an aggressively simple language, and your search engine will point you to dozens of C tutorials online; if you have experience with another language, you should be able to get the basics in an hour or two.

I might as well point out to you that I have also written a textbook on statistical and scientific computing, *Modeling with Data* [Klemens 2008]. Along with lots of details of dealing with numeric data and using statistical models for describing data, it has a standalone tutorial on C, which I naturally think overcomes many of the failings of older C tutorials.

Q: I'm an application programmer, not a kernel hacker. Why should I use C instead of a quick-to-write scripting language like Python?

A: If you are an application programmer, this book is for you. I read people asserting that C is a systems language, which impresses me as so un-punk—who are they to tell us what we're allowed to write?

Statements in the way of "Our language is almost as fast as C, but is easier to write" are so common that they are almost cliché. Well, C is definitely as fast as C, and the purpose of this book is to show you that C is easier to write than the textbooks from decades past imply that it is. You don't have to call `malloc` and get elbow-deep in memory management half as often as the systems programmers of the 1990s did, we have facilities for easier string handling, and even the core syntax has evolved to make for more legible code.

I started writing C in earnest because I had to speed up a simulation in a scripting language, R. Like so many other scripting languages, R has a C interface and encourages the user to make use of it any time the host language is too slow. Eventually, I had so many functions jumping out from the R script to C code that I just dropped the host language entirely. Next thing you know, I'm writing a book on modern C technique.

Q: It's nice that application programmers coming from scripting languages will like this book, but I *am* a kernel hacker. I taught myself C in fifth grade and sometimes have dreams that correctly compile. What new material can there be for me?

A: C really has evolved in the last 20 years. As I'll discuss later, the set of things we are guaranteed that all C compilers support has changed with time, thanks to two new C

standards since the ANSI standard that defined the language for so long. Maybe have a look at Chapter 10 and see if anything there surprises you.

Also, the environment has advanced. Autotools has entirely changed how distribution of code happens, meaning that it is much easier to reliably call other libraries, meaning that our code should spend less time reinventing common structures and routines and more time calling the sort of libraries discussed throughout this book.

Q: I can't help but notice that about a third of this book has almost no C code in it.

A: It is true: good C practice requires gathering good C tools. If you're not using a debugger (standalone or part of your IDE), you're making your life much more difficult. If you tell me that it's impossible to track down memory leaks, then that means that you haven't heard of Valgrind, a system designed to point out the exact line where memory leaks and errors occurred. Python and company have built-in package managers; C's *de facto* cross-platform packaging system, Autotools, is a standalone system that is its own story.

If you use an attractive Integrated Development Environment (IDE) as a wrapper for all these various tools, you may still benefit from knowing how your IDE is dealing with environment variables and other minutiæ that have been hidden from you but still crop up and throw errors at you.

Q: Some of these tools you talk about are *old*. Aren't there more modern alternatives to these shell-oriented tools?

A: If we make fun of people who reject new things just because they're new, then we have no right to reject old things just because they're old.

One can find reasonable sources putting the first six-string guitar around 1200, the first four-string violin circa 1550, and the piano with keyboard around 1700. The odds are good that most (if not all) of the music you listen to today will involve one of these instruments. Punk rock didn't happen by rejecting the guitar, but by using it creatively, such as piping the guitar's output through new filters.

Q: I have the Internet, and can look up commands and syntax details in a second or two, so, really, why should I read this book?

A: It's true: you can get an operator precedence table from a Linux or Mac command prompt with `man operator`, so why am I going to put one here?

I've got the same Internet you've got, and I've spent a lot of time reading it. So I have a good idea of what isn't being talked about, and that's what I stick to here. When introducing a new tool, like gprof or GDB, I give you what you need to know to get your bearings and ask your search engine coherent questions, and what other textbooks missed (which is a lot).

Standards: So Many to Choose From

Unless explicitly stated otherwise, everything in this book conforms to the ISO C99 and C11 standards. To make sense of what that means, and give you some historical background, let us go through the list of major C standards (passing on the minor revisions and corrections).

K & R (circa 1978)
: Dennis Ritchie, Ken Thompson, and a handful of other contributors came up with C while putting together the Unix operating system. Brian Kernighan and Dennis Ritchie eventually wrote down a description of the language in the first edition of their book, which set the first *de facto* standard [Kernighan 1978].

ANSI C89
: Bell Labs handed over the stewardship of the language to the American National Standards Institute. In 1989, they published their standard, which made a few improvements over K & R. The second edition of K & R's book included a full specification of the language, which meant that tens of thousands of programmers had a copy of the ANSI standard on their desks [Kernighan 1988]. The ANSI standard was adopted by the ISO in 1990 with no serious changes, but *ANSI '89* seems to be the more common term (and would make a great t-shirt slogan).

A decade passed. C went mainstream, in the sense that the base code for more or less every PC and every Internet server was written in C, which is as mainstream as a human endeavor could possibly become.

During this period, C++ split off and hit it big (although not quite as big). C++ was the best thing to ever happen to C. While every other language was bolting on extra syntax to follow the object-oriented trend and whatever other new tricks came to the authors' minds, C stuck to the standard. The people who wanted stability and portability used C, the people who wanted more and more features so they could wallow in them like moist hundred dollar bills got C++, and everybody was happy.

ISO C99
: The C standard underwent a major revision a decade later. Additions were made for numeric and scientific computing, with a standard type for complex numbers and some type-generic functions. A few conveniences from C++ got lifted, including one-line comments (which originally came from one of C's predecessor languages, BCPL) and being able to declare variables at the head of `for` loops. Using structures was made easier thanks to a few additions to the rules for how they can be declared and initialized, plus some notational conveniences. Things were modernized to acknowledge that security matters and that not everybody speaks English.

  When you think about just how much of an impact C89 had, and how the entire globe was running on C code, it's hard to imagine the ISO being able to put out anything that wouldn't be widely criticized—even a refusal to make any changes

would be reviled. And indeed, this standard was controversial. There are two common ways to express a complex variable (rectangular and polar coordinates)—so where does the ISO get off picking one? Why do we need a mechanism for variable-length macro inputs when all the good code got written without it? In other words, the purists accused the ISO of selling out to the pressure for more features.

As of this writing, most compilers support C99 plus or minus a few caveats; the `long double` type seems to cause a lot of trouble, for example. However, there is one notable exception to this broad consensus: Microsoft currently refuses to add C99 support to its Visual Studio C++ compiler. The section "Compiling C with Windows" on page 6 covers some of the many ways to compile C code for Windows, so not using Visual Studio is at most an inconvenience, and having a major establishment player tell us that we can't use ISO-standard C only bolsters the punk rock of it all.

C11

Self-conscious about the accusations of selling out, the ISO made few serious changes in the third edition of the standard. We got a means of writing type-generic functions, and things were modernized to further acknowledge that security matters and that not everybody speaks English.

I'm writing this in 2012, shortly after the C11 standard came out in December of 2011, and there's already some support from compilers and libraries.

The POSIX Standard

That's the state of things as far as C itself goes, but the language coevolved with the Unix operating system, and you will see throughout the book that the interrelationship matters for day-to-day work. If something is easy on the Unix command line, then it is probably because it is easy in C; Unix tools are often written to facilitate writing C code.

Unix

C and Unix were designed at Bell Labs in the early 1970s. During most of the 20th century, Bell was being investigated for monopolistic practices, and one of its agreements with the US federal government included promises that Bell would not expand its reach into software. So Unix was given away for free for researchers to dissect and rebuild. The name Unix is a trademark, originally owned by Bell Labs and subsequently traded off like a baseball card among a number of companies.

Variants of Unix blossomed, as the code was looked over, reimplemented, and improved in different ways by diverse hackers. It just takes one little incompatibility to make a program or script unportable, so the need for some standardization quickly became apparent.

POSIX

This standard, first established by the Institute of Electrical and Electronics Engineers (IEEE) in 1988, provided a common basis for Unix-like operating systems. It specifies how the shell should work, what to expect from commands like `ls` and `grep`, and a number of C libraries that C authors can expect to have available. For example, the pipes that command-line users use to string together commands are specified in detail here, which means C's `popen` (pipe open) function is POSIX-standard, not ISO C standard. The POSIX standard has been revised many times; the version as of this writing is POSIX:2008, and is what I am referring to when I say that something is POSIX-standard. A POSIX-standard system must have a C compiler available, via the command name `c99`.

This book will make use of the POSIX standard, though I'll tell you when.

With the exception of many members of a family of OSes from Microsoft, just about every current operating system you could name is built on a POSIX-compliant base: Linux, Mac OS X, iOS, webOS, Solaris, BSD—even Windows servers offer a POSIX subsystem. And for the hold-out OSes, "Compiling C with Windows" on page 6 will show you how to install a POSIX subsystem.

Finally, there are two more implementations of POSIX worth noting because of their prevalence and influence:

BSD

After Unix was sent out from Bell Labs for researchers to dissect, the nice people at the University of California, Berkeley, made major improvements, eventually rewriting the entire Unix code base to produce the Berkeley Software Distribution. If you are using a computer from Apple, Inc., you are using BSD with an attractive graphical frontend. BSD goes beyond POSIX in several respects, and we'll see a function or two that are not part of the POSIX standard but are too useful to pass up (most notably the lifesaver that is `asprintf`).

GNU

It stands for GNU's Not Unix, and is the other big success story in independently reimplementing and improving on the Unix environment. The great majority of Linux distributions use GNU tools throughout. There are very good odds that you have the GNU Compiler Collection (gcc) on your POSIX box—even BSD uses it. Again, the gcc defines a *de facto* standard that extends C and POSIX in a few ways, and I will be explicit when making use of those extensions.

Legally, the BSD license is slightly more permissive than the GNU license. Because some parties are deeply concerned with the political and business implications of the licenses, one can typically find both GNU and BSD versions of most tools. For example, both the GNU Compiler Collection (`gcc`) and the BSD's `clang` are top-notch C compilers. The authors from both camps closely watch and learn from each other's work, so we can expect that the differences that currently exist will tend to even out over time.

The Legal Sidebar

US law no longer has a registration system for copyright: with few exceptions, as soon as anybody writes something down, it is copyrighted.

Of course, distribution of a library depends on copying from hard drive to hard drive, and there are a number of common mechanisms for granting the right to copy a copyrighted work with little hassle.

- The GNU Public License allows unlimited copying and use of the source code and its executable version. There is one major condition: If you *distribute* a program or library based on the GPLed source code, then you must distribute the source code to your program. Note well that if you use your program in-house and don't distribute it, this condition doesn't hold, and you have no obligation to distribute source. Running a GPLed program, like compiling your code with gcc, does not in itself obligate you to distribute source code, because the program output (such as the executable you just compiled) is not considered to be based on or a derivative of gcc. [Example: the GNU Scientific Library.]
- The Lesser GPL is much like the GPL, but it explicitly stipulates that if you are linking to an LGPL library as a shared library, then your code doesn't count as a derivative work, and you aren't obligated to distribute source. That is, you can distribute closed-source code that links to an LGPL library. [Example: GLib.]
- The BSD license requires that you preserve copyrights and disclaimers for BSD-licensed source code, but doesn't require that you redistribute source code. [Example: Libxml2, under the BSD-like MIT license.]

Please note the usual disclaimer: I am not a lawyer, and this is a sidebar summary of several rather long legal documents. Read the documents themselves or consult a lawyer if you are unsure about how the details apply to your situation.

Some Logistics

Conventions Used in This Book

The following typographical conventions are used in this book:

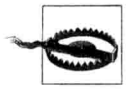
Italic
: Indicates new terms, filenames and file paths, URLs, and email addresses. Many new terms are defined in a glossary at the end of this book.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width italic
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.

Your Turn: These are exercises, to help you learn by doing and give you an excuse to get your hands on a keyboard.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

The code examples for this title can be found here: *http://examples.oreilly.com/0636920025108/*.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*21st Century C* by Ben Klemens (O'Reilly). Copyright 2013 Ben Klemens, 978-1-449-32714-9."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online (*www.safaribooksonline.com*) is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://oreil.ly/21st_century_c*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

Nora Albert: general support, guinea pig.
Bruce Fields, Dave Kitabjian, Sarah Weissman: extensive and thorough review.
Patrick Hall: Unicode erudition.
Nathan Jepson and Shawn Wallace: editorial.
Rolando Rodríguez: testing, inquisitive use, and exploration.
Rachel Steely: production.
Ulrik Sverdrup: pointing out that we can use repeated designated initializers to set default values.

Table of Contents

Part II. The Language