

FOREWORDS BY WALTER BRIGHT AND SCOTT MEYERS



The D Programming Language



Andrei Alexandrescu

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: informat.com/aw

Library of Congress Cataloging-in-Publication Data

Alexandrescu, Andrei.

The D Programming Language / Andrei Alexandrescu.
p. cm.

Includes bibliographical references and index.

ISBN 978-0-321-63536-5 (pbk. : alk. paper) 1. D (Computer program language) I. Title.

QA76.73.D138A44 2010

005.13'3—dc22

2010009924

Copyright © 2010 Pearson Education

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671-3447

ISBN 13: 978-0-321-63536-5

ISBN 10: 0-321-63536-1

Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.
First printing, May 2010

Contents

Foreword by Walter Bright	xv
Foreword by Scott Meyers	xix
Preface	xxiii
Intended Audience	xxiv
Organization of the Book	xxiv
A Brief History	xxv
Acknowledgments	xxvi
1 “D”iving In	1
1.1 Numbers and Expressions	3
1.2 Statements	5
1.3 Function Basics	6
1.4 Arrays and Associative Arrays	7
1.4.1 Building a Vocabulary	7
1.4.2 Array Slicing. Type-Generic Functions. Unit Tests	10
1.4.3 Counting Frequencies. Lambda Functions	12
1.5 Basic Data Structures	14
1.6 Interfaces and Classes	20
1.6.1 More Statistics. Inheritance	23
1.7 Values versus References	25
1.8 Summary	27
2 Basic Types. Expressions	29
2.1 Symbols	30
2.1.1 Special Symbols	31
2.2 Literals	32
2.2.1 Boolean Literals	32
2.2.2 Integral Literals	32
2.2.3 Floating-Point Literals	33
2.2.4 Character Literals	34
2.2.5 String Literals	35

2.2.6	Array and Associative Array Literals	39
2.2.7	Function Literals	40
2.3	Operators	42
2.3.1	Lvalues and Rvalues	42
2.3.2	Implicit Numeric Conversions	42
2.3.3	Typing of Numeric Operators	45
2.3.4	Primary Expressions	46
2.3.5	Postfix Expressions	49
2.3.6	Unary Expressions	51
2.3.7	The Power Expression	54
2.3.8	Multiplicative Expressions	54
2.3.9	Additive Expressions	55
2.3.10	Shift Expressions	55
2.3.11	in Expressions	56
2.3.12	Comparison Operators	56
2.3.13	Bitwise OR, XOR, AND	58
2.3.14	Logical AND	59
2.3.15	Logical OR	59
2.3.16	The Conditional Operator	59
2.3.17	Assignment Operators	60
2.3.18	The Comma Operator	60
2.4	Summary and Quick Reference	61
3	Statements	65
3.1	The Expression Statement	65
3.2	The Compound Statement	66
3.3	The if Statement	67
3.4	The static if Statement	68
3.5	The switch Statement	71
3.6	The final switch Statement	72
3.7	Looping Statements	73
3.7.1	The while Statement	73
3.7.2	The do-while Statement	73
3.7.3	The for Statement	74
3.7.4	The foreach Statement	74
3.7.5	foreach on Arrays	75
3.7.6	The continue and break Statements	78
3.8	The goto Statement	78
3.9	The with Statement	80
3.10	The return Statement	81
3.11	The throw and try Statements	81
3.12	The mixin Statement	82

3.13	The scope Statement	84
3.14	The synchronized Statement	88
3.15	The asm Statement	89
3.16	Summary and Quick Reference	89
4	Arrays, Associative Arrays, and Strings	93
4.1	Dynamic Arrays	93
4.1.1	Length	95
4.1.2	Bounds Checking	95
4.1.3	Slicing	97
4.1.4	Copying	98
4.1.5	Comparing for Equality	100
4.1.6	Concatenating	100
4.1.7	Array-wise Expressions	100
4.1.8	Shrinking	102
4.1.9	Expanding	103
4.1.10	Assigning to <code>.length</code>	106
4.2	Fixed-Size Arrays	107
4.2.1	Length	108
4.2.2	Bounds Checking	108
4.2.3	Slicing	109
4.2.4	Copying and Implicit Conversion	109
4.2.5	Comparing for Equality	110
4.2.6	Concatenating	111
4.2.7	Array-wise Operations	111
4.3	Multidimensional Arrays	111
4.4	Associative Arrays	114
4.4.1	Length	114
4.4.2	Reading and Writing Slots	115
4.4.3	Copying	115
4.4.4	Comparing for Equality	116
4.4.5	Removing Elements	116
4.4.6	Iterating	116
4.4.7	User-Defined Types as Keys	117
4.5	Strings	118
4.5.1	Code Points	118
4.5.2	Encodings	119
4.5.3	Character Types	120
4.5.4	Arrays of Characters + Benefits = Strings	121
4.6	Arrays' Maverick Cousin: The Pointer	124
4.7	Summary and Quick Reference	126

5 Data and Functions. Functional Style	131
5.1 Writing and unit testing a Simple Function	131
5.2 Passing Conventions and Storage Classes	134
5.2.1 ref Parameters and Returns	135
5.2.2 in Parameters	135
5.2.3 out Parameters	136
5.2.4 static Data	137
5.3 Type Parameters	138
5.4 Signature Constraints	140
5.5 Overloading	142
5.5.1 Partial Ordering of Functions	144
5.5.2 Cross-Module Overloading	146
5.6 Higher-Order Functions. Function Literals	148
5.6.1 Function Literals versus Delegate Literals	150
5.7 Nested Functions	150
5.8 Closures	152
5.8.1 OK, This Works. Wait, It Shouldn't. Oh, It Does!	154
5.9 Beyond Arrays. Ranges. Pseudo Members	154
5.9.1 Pseudo Members and the @property Attribute	156
5.9.2 reduce—Just Not <i>ad Absurdum</i>	157
5.10 Variadic Functions	159
5.10.1 Homogeneous Variadic Functions	159
5.10.2 Heterogeneous Variadic Functions	160
5.11 Function Attributes	165
5.11.1 Pure Functions	165
5.11.2 The nothrow Function Attribute	168
5.12 Compile-Time Evaluation	169
6 Classes. Object-Oriented Style	175
6.1 Classes	175
6.2 Object Names Are References	177
6.3 It's an Object's Life	181
6.3.1 Constructors	181
6.3.2 Forwarding Constructors	183
6.3.3 Construction Sequence	184
6.3.4 Destruction and Deallocation	186
6.3.5 Tear-Down Sequence	187
6.3.6 Static Constructors and Destructors	188
6.4 Methods and Inheritance	190
6.4.1 A Terminological Smörgåsbord	191
6.4.2 Inheritance Is Subtyping. Static and Dynamic Type	192
6.4.3 Overriding Is Only Voluntary	193

6.4.4	Calling Overridden Methods	194
6.4.5	Covariant Return Types	195
6.5	Class-Level Encapsulation with <code>static</code> Members	196
6.6	Curbing Extensibility with <code>final</code> Methods	197
6.6.1	<code>final</code> Classes	199
6.7	Encapsulation	199
6.7.1	<code>private</code>	200
6.7.2	<code>package</code>	200
6.7.3	<code>protected</code>	200
6.7.4	<code>public</code>	201
6.7.5	<code>export</code>	201
6.7.6	How Much Encapsulation?	201
6.8	One Root to Rule Them All	203
6.8.1	<code>string</code> <code>toString()</code>	205
6.8.2	<code>size_t</code> <code>toHash()</code>	205
6.8.3	<code>bool</code> <code>opEquals(Object rhs)</code>	205
6.8.4	<code>int</code> <code>opCmp(Object rhs)</code>	209
6.8.5	<code>static Object</code> <code>factory(string className)</code>	210
6.9	Interfaces	212
6.9.1	The Non-Virtual Interface (NVI) Idiom	213
6.9.2	<code>protected</code> Primitives	216
6.9.3	Selective Implementation	217
6.10	Abstract Classes	218
6.11	Nested Classes	222
6.11.1	Classes Nested in Functions	223
6.11.2	<code>static</code> Nested Classes	225
6.11.3	Anonymous Classes	226
6.12	Multiple Inheritance	226
6.13	Multiple Subtyping	230
6.13.1	Overriding Methods in Multiple Subtyping Scenarios	231
6.14	Parameterized Classes and Interfaces	233
6.14.1	Heterogeneous Translation, Again	235
6.15	Summary	237
7	Other User-Defined Types	239
7.1	<code>structs</code>	240
7.1.1	Copy Semantics	241
7.1.2	Passing <code>struct</code> Objects to Functions	242
7.1.3	Life Cycle of a <code>struct</code> Object	243
7.1.4	Static Constructors and Destructors	254
7.1.5	Methods	255
7.1.6	<code>static</code> Members	260

7.1.7	Access Specifiers	261
7.1.8	Nesting <code>structs</code> and <code>classes</code>	261
7.1.9	Nesting <code>structs</code> inside Functions	262
7.1.10	Subtyping with <code>structs</code> . The <code>@disable</code> Attribute	263
7.1.11	Field Layout, Alignment	266
7.2	<code>unions</code>	270
7.3	Enumerated Values	272
7.3.1	Enumerated Types	274
7.3.2	<code>enum</code> Properties	275
7.4	<code>alias</code>	276
7.5	Parameterized Scopes with <code>template</code>	278
7.5.1	Eponymous templates	281
7.6	Injecting Code with <code>mixin</code> templates	282
7.6.1	Symbol Lookup inside a <code>mixin</code>	284
7.7	Summary and Reference	285
8	Type Qualifiers	287
8.1	The <code>immutable</code> Qualifier	288
8.1.1	Transitivity	289
8.2	Composing with <code>immutable</code>	291
8.3	<code>immutable</code> Parameters and Methods	292
8.4	<code>immutable</code> Constructors	293
8.5	Conversions involving <code>immutable</code>	295
8.6	The <code>const</code> Qualifier	297
8.7	Interaction between <code>const</code> and <code>immutable</code>	298
8.8	Propagating a Qualifier from Parameter to Result	299
8.9	Summary	300
9	Error Handling	301
9.1	<code>throwing</code> and <code>catching</code>	301
9.2	Types	302
9.3	<code>finally</code> clauses	306
9.4	<code>nothrow</code> Functions and the Special Nature of <code>Throwable</code>	307
9.5	Collateral Exceptions	307
9.6	Stack Unwinding and Exception-Safe Code	309
9.7	Uncaught Exceptions	312
10	Contract Programming	313
10.1	Contracts	314
10.2	Assertions	316
10.3	Preconditions	317
10.4	Postconditions	319

10.5	Invariants	321
10.6	Skipping Contract Checks. Release Builds	324
10.6.1	enforce Is Not (Quite) assert	325
10.6.2	assert(false)	326
10.7	Contracts: Not for Scrubbing Input	327
10.8	Contracts and Inheritance	329
10.8.1	Inheritance and in Contracts	330
10.8.2	Inheritance and out Contracts	332
10.8.3	Inheritance and invariant Contracts	334
10.9	Contracts in Interfaces	334
11	Scaling Up	337
11.1	Packages and Modules	337
11.1.1	import Declarations	338
11.1.2	Module Searching Roots	340
11.1.3	Name Lookup	341
11.1.4	public import Declarations	344
11.1.5	static import Declarations	345
11.1.6	Selective imports	346
11.1.7	Renaming in imports	347
11.1.8	The module Declaration	348
11.1.9	Module Summaries	349
11.2	Safety	353
11.2.1	Defined and Undefined Behavior	354
11.2.2	The @safe, @trusted, and @system Attributes	355
11.3	Module Constructors and Destructors	356
11.3.1	Execution Order within a Module	357
11.3.2	Execution Order across Modules	358
11.4	Documentation Comments	358
11.5	Interfacing with C and C++	359
11.6	deprecated	359
11.7	version Declarations	360
11.8	debug Declarations	361
11.9	D's Standard Library	361
12	Operator Overloading	365
12.1	Overloading Operators	366
12.2	Overloading Unary Operators	367
12.2.1	Using mixin to Consolidate Operator Definitions	368
12.2.2	Postincrement and Postdecrement	369
12.2.3	Overloading the cast Operator	369
12.2.4	Overloading Ternary Operator Tests and if Tests	370

12.3	Overloading Binary Operators	371
12.3.1	Operator Overloading ²	373
12.3.2	Commutativity	373
12.4	Overloading Comparison Operators	375
12.5	Overloading Assignment Operators	376
12.6	Overloading Indexing Operators	377
12.7	Overloading Slicing Operators	379
12.8	The \$ Operator	379
12.9	Overloading <code>foreach</code>	380
12.9.1	<code>foreach</code> with Iteration Primitives	380
12.9.2	<code>foreach</code> with Internal Iteration	381
12.10	Defining Overloaded Operators in Classes	383
12.11	And Now for Something Completely Different: <code>opDispatch</code>	384
12.11.1	Dynamic Dispatch with <code>opDispatch</code>	386
12.12	Summary and Quick Reference	388
13	Concurrency	391
13.1	Concurrentgate	392
13.2	A Brief History of Data Sharing	394
13.3	Look, Ma, No (Default) Sharing	397
13.4	Starting a Thread	399
13.4.1	<code>immutable</code> Sharing	400
13.5	Exchanging Messages between Threads	401
13.6	Pattern Matching with <code>receive</code>	403
13.6.1	First Match	405
13.6.2	Matching Any Message	405
13.7	File Copying—with a Twist	406
13.8	Thread Termination	407
13.9	Out-of-Band Communication	409
13.10	Mailbox Crowding	410
13.11	The <code>shared</code> Type Qualifier	411
13.11.1	The Plot Thickens: <code>shared</code> Is Transitive	412
13.12	Operations with <code>shared</code> Data and Their Effects	413
13.12.1	Sequential Consistency of <code>shared</code> Data	414
13.13	Lock-Based Synchronization with <code>synchronized classes</code>	414
13.14	Field Typing in <code>synchronized classes</code>	419
13.14.1	Temporary Protection == No Escape	419
13.14.2	Local Protection == Tail Sharing	420
13.14.3	Forcing Identical Mutexes	422
13.14.4	The Unthinkable: casting Away <code>shared</code>	423
13.15	Deadlocks and the <code>synchronized</code> Statement	424
13.16	Lock-Free Coding with <code>shared classes</code>	426

13.16.1 shared classes	427
13.16.2 A Couple of Lock-Free Structures	427
13.17 Summary	431
Bibliography	433
Index	439

This page intentionally left blank

Foreword

by Walter Bright

There's a line in a science fiction novel I read long ago that says a scientist would fearlessly peer into the gates of hell if he thought it would further knowledge in his field. In one sentence, it captures the essence of what it means to be a scientist. This joy in discovery, this need to know, is readily apparent in the videos and writings of physicist Richard Feynman, and his enthusiasm is infectious and enthralling.

Although I am not a scientist, I understand their motivation. Mine is that of an engineer—the joy of creation, of building something out of nothing. One of my favorite books is a chronicle of the step-by-step process the Wright brothers went through to solve the problems of flight one by one, *The Wright Brothers as Engineers* by Wald, and how they poured all that knowledge into creating a flying machine.

My early interests were summed up in the opening pages of *Rocket Manual for Amateurs* by Brinley with the phrase “thrilled and fascinated by things that burn and explode,” later matured into wanting to build things that went faster and higher.

But building powerful machines is an expensive proposition. And then I discovered computers. The marvelous and seductive thing about computers is the ease with which things can be built. You don't need a billion-dollar fab plant, a machine shop, or even a screwdriver. With just an inexpensive computer, you can create worlds.

So I started creating imaginary worlds on the computer. The first was the game Empire, Wargame of the Century. The computers of the day were too underpowered to run it properly, so I became interested in how to optimize the performance of programs. This led to studying the compilers that generated the code and naturally to the hubris of “I can write a better compiler than that.” Enamored with C, I gravitated toward implementing a C compiler. That wasn't too hard, taking a couple of years part-time. Then I discovered Bjarne Stroustrup's C++ language, and I thought that I could add those extensions to the C compiler in a couple of months (!).

Over a decade later, I was still working on it. In the process of implementing it, I became very familiar with every detail of the language. Supporting a large user base meant a lot of experience in how other people perceived the language, what worked, and what didn't. I'm not able to use something without thinking of ways to improve the design. In 1999, I decided to put this into practice. It started out as the Mars programming lan-

guage, but my colleagues called it D first as a joke, but the name caught on and the D programming language was born.

D is ten years old as of this writing and has produced its second major incarnation, sometimes called D2. In that time D has expanded from one man toiling over a keyboard to a worldwide community of developers working on all facets of the language and supporting an ecosystem of libraries and tools.

The language itself (which is the focus of this book) has grown from modest beginnings to a very powerful language adept at solving programming problems from many angles. To the best of my knowledge, D offers an unprecedentedly adroit integration of several powerful programming paradigms: imperative, object-oriented, functional, and meta.

At first blush, it would appear that such a language could not be simple. And indeed, D is not a simple language. But I'd argue that is the wrong way to view a language. A more useful view is, what do programming solutions in that language look like? Are D programs complicated and obtuse, or simple and elegant?

A colleague of mine who has extensive experience in a corporate environment observed that an IDE (Integrated Development Environment) was an essential tool for programming because with one click a hundred lines of boilerplate code could be generated. An IDE is not as essential a tool for D, because instead of relying on wizard-based boilerplate generation, D obviates the boilerplate itself by using introspection and generational capabilities. The programmer doesn't have to see that boilerplate. The inherent complexity of the program is taken care of by the language, rather than an IDE.

For example, suppose one wanted to do OOP (object-oriented programming) using a simpler language that has no particular support for the paradigm. It can be done, but it's just awful and rarely worthwhile. But when a more complex language supports OOP directly, then writing OOP programs becomes simple and elegant. The language is more complicated, but the user code is simpler. This is worthwhile progress.

The ability to write user code for a wide variety of tasks in a simple and elegant manner pretty much requires a language that supports multiple programming paradigms. Properly written code should just look beautiful on the page, and beautiful code oddly enough tends to be correct code. I'm not sure why that relationship holds, but it tends to be true. It's the same way an airplane that looks good tends to fly well, too. Therefore, language features that enable algorithms to be expressed in a beautiful way are probably good things.

Simplicity and elegance in writing code, however, are not the only metrics that characterize a good programming language. These days, programs are rapidly increasing in size with no conceivable end in sight. With such size, it becomes less and less practical to rely on convention and programming expertise to ensure the code is correct, and more and more worthwhile to rely on machine-checkable guarantees. To that end, D sports a variety of strategies that the programmer can employ to make such guarantees. These include contracts, memory safety, various function attributes, immutability, hijack protection, scope guards, purity, unit tests, and thread data isolation.

No, we haven't overlooked performance! Despite many predictions that performance is no longer relevant, despite computers running a thousand times faster than when I wrote my first compiler, there never seems to be any shortage of demand for faster programs. D is a systems programming language. What does that mean? In one sense, it means that one can write an operating system in D, as well as device drivers and application code. In a more technical sense, it means that D programs have access to all the capabilities of the machine. This means you can use pointers, do pointer aliasing and pointer arithmetic, bypass the type system, and even write code directly in assembly language. There is nothing completely sealed off from a D programmer's access. For example, the implementation of D's garbage collector is entirely written in D.

But wait! How can that be? How can a language offer both soundness guarantees and arbitrary pointer manipulation? The answer is that the kinds of guarantees are based on the language constructs used. For example, function attributes and type constructors can be used to state guarantees enforceable at compile time. Contracts and invariants specify guarantees to be enforced at runtime.

Most of D's features have appeared in other languages in one form or another. Any particular one doesn't make the case for a language. But the combination is more than the sum of the parts, and D's combination makes for a satisfying language that has elegant and straightforward means to solve an unusually wide variety of programming problems.

Andrei Alexandrescu is famous for his unconventional programming ideas becoming the new mainstream (see his seminal book *Modern C++ Design*). Andrei joined the D programming language design team in 2006. He's brought with him a sound theoretical grounding in programming, coupled with an endless stream of innovative solutions to programming design problems. Much of the shape of D2 is due to his contributions, and in many ways this book has co-evolved with D. One thing you'll happily discover in his writing about D is the *why* of the design choices, rather than just a dry recitation of facts. Knowing why a language is the way it is makes it much easier and faster to understand and get up to speed.

Andrei goes on to illustrate the whys by using D to solve many fundamental programming problems. Thus he shows not only how D works, but why it works, and how to use it.

I hope you'll have as much fun programming in D as I've had working to bring it to life. A palpable excitement about the language seeps out of the pages of Andrei's book. I think you'll find it exciting!

Walter Bright
January 2010

This page intentionally left blank

Foreword

by Scott Meyers

By any measure, C++ has been a tremendous success, but even its most ardent proponents won't deny that it's a complicated beast. This complexity influenced the design of C++'s most widely used successors, Java and C#. Both strove to avoid C++'s complexity—to provide most of its functionality in an easier-to-use package.

Complexity reduction took two basic forms. One was elimination of "complicated" language features. C++'s need for manual memory management, for example, was obviated by garbage collection. Templates were deemed to fail the cost/benefit test, so the initial versions of these languages chose to exclude anything akin to C++'s support for generics.

The other form of complexity reduction involved replacing "complicated" C++ features with similar, but less demanding, constructs. C++'s multiple inheritance morphed into single inheritance augmented with interfaces. Current versions of Java and C# support templatesque generics, but they're simpler than C++'s templates.

These successor languages aspired to far more than simply doing what C++ did with reduced complexity. Both defined virtual machines, added support for runtime reflection, and provided extensive libraries that allow many programmers to shift their focus from creating new code to gluing existing components together. The result can be thought of as C-based "productivity languages." If you want to quickly create software that more or less corresponds to combinations of existing components—and much software falls into this category—Java and C# are better choices than C++.

But C++ isn't a productivity language; it's a *systems* programming language. It was designed to rival C in its ability to communicate with hardware (e.g., in drivers and embedded systems), to work with C-based libraries and data structures without adaptation (e.g., in legacy systems), to squeeze the last drop of performance out of the hardware it runs on. It's not really an irony that the performance-critical components of the virtual machines beneath Java and C# are written in C++. The high-performance implementation of virtual machines is a job for a *systems* language, not a productivity language.

D aims to be C++'s successor in the realm of systems programming. Like Java and C#, D aims to avoid the complexity of C++, and to this end it uses some of the same

techniques. Garbage collection is in, manual memory management is out.¹ Single inheritance and interfaces are in, multiple inheritance is out. But then D starts down a path of its own.

It begins by identifying functional holes in C++ and filling them. Current C++ offers no Unicode support, and its nascent successor version (C++0x) provides only a limited amount. D handles Unicode from the get-go. Neither current C++ nor C++0x offers support for modules, Contract Programming, unit testing, or “safe” subsets (where memory errors are impossible). D offers all these things, and it does so without sacrificing the ability to generate high-quality native code.

Where C++ is both powerful and complicated, D aims to be at least as powerful but less complicated. Template metaprogrammers in C++ have demonstrated that compile-time computation is an important technology, but they’ve had to jump through hoops of syntactic fire to practice it. D offers similar capabilities, but without the lexical pain. If you know how to write a function in current C++, you know nothing about how to write the corresponding C++ function that’s evaluated during compilation. If you know how to write a function in D, however, you know exactly how to write its compile-time variant, because the code is the same.

One of the most interesting places where D parts ways with its C++-derived siblings is in its approach to thread-based concurrency. Recognizing that improperly synchronized access to shared data (data races) is a pit that’s both easy to fall into and hard to climb out of, D turns convention on its head: by default, data isn’t shared across threads. As D’s designers point out, given the deep cache hierarchies of modern hardware, memory often isn’t truly shared across cores or processors anyway, so why default to offering developers an abstraction that’s not only an illusion, it’s an illusion known to facilitate the introduction of difficult-to-debug errors?

All these things and more make D a noteworthy point in the C heritage design space, and that is reason enough to read this book. The fact that the author is Andrei Alexandrescu makes the case even stronger. As codesigner of D and an implementer of substantial portions of its library, Andrei knows D like almost no one else. Naturally, he can describe the D programming language, but he can also explain *why* D is the way it is. Features present in the language are there for a reason, and would-be features that are missing are absent for a reason, too. Andrei is in a unique position to illuminate such reasoning.

This illumination comes through in a uniquely engaging style. In the midst of what might seem to be a needless digression (but is actually a waystation en route to a destination he needs you to reach), Andrei offers reassurance: “I know you are asking yourself what this has to do with compile-time evaluation. It does. Please bear with me.” Regarding the unintuitive nature of linker diagnostics, Andrei observes, “If you forget about `--main`, don’t worry; the linker will fluently and baroquely remind you of that

1. Actually, it’s optional. As befits a systems programming language, if you really want to perform manual memory management, D will let you.

in its native language, encrypted Klingon.” Even references to other publications get the Alexandrescu touch. You’re not simply referred to Wadler’s “Proofs are programs,” you’re referred to “Wadler’s fascinating monograph ‘Proofs are programs.’” Friedl’s “Mastering regular expressions” isn’t just recommended, it’s “warmly recommended.”

A book about a programming language is filled with sample code, of course, and the code samples also demonstrate that Andrei is anything but a pedestrian author. Here’s his prototype for a search function:

```
bool find(int[] haystack, int needle);
```

This is a book by a skilled author describing an interesting programming language. I’m sure you’ll find the read rewarding.

Scott Meyers
January 2010

This page intentionally left blank

Preface

Programming language design seeks power in simplicity and, when successful, begets beauty.

Choosing the trade-offs among contradictory requirements is a difficult task that requires good taste from the language designer as much as mastery of theoretical principles and of practical implementation matters. Programming language design is software-engineering-complete.

D is a language that attempts to consistently do the right thing within the constraints it chose: system-level access to computing resources, high performance, and syntactic similarity with C-derived languages. In trying to do the right thing, D sometimes stays with tradition and does what other languages do, and other times it breaks tradition with a fresh, innovative solution. On occasion that meant revisiting the very constraints that D ostensibly embraced. For example, large program fragments or indeed entire programs can be written in a well-defined memory-safe subset of D, which entails giving away a small amount of system-level access for a large gain in program debuggability.

You may be interested in D if the following values are important to you:

- *Performance.* D is a systems programming language. It has a memory model that, although highly structured, is compatible with C's and can call into and be called from C functions without any intervening translation.
- *Expressiveness.* D is not a small, minimalistic language, but it does have a high power-to-weight ratio. You can define eloquent, self-explanatory designs in D that model intricate realities accurately.
- *“Torque.”* Any backyard hot-rodder would tell you that power isn't everything; its availability is. Some languages are most powerful for small programs, whereas other languages justify their syntactic overhead only past a certain size. D helps you get work done in short scripts and large programs alike, and it isn't unusual for a large program to grow organically from a simple single-file script.
- *Concurrency.* D's approach to concurrency is a definite departure from the languages it resembles, mirroring the departure of modern hardware designs from the architectures of yesteryear. D breaks away from the curse of implicit memory sharing (though it allows statically checked explicit sharing) and fosters mostly independent threads that communicate with one another via messages.

- *Generic code.* Generic code that manipulates other code has been pioneered by the powerful Lisp macros and continued by C++ templates, Java generics, and similar features in various other languages. D offers extremely powerful generic and generational mechanisms.
- *Eclecticism.* D recognizes that different programming paradigms are advantageous for different design challenges and fosters a highly integrated federation of styles instead of One True Approach.
- *“These are my principles. If you don’t like them, I’ve got others.”* D tries to observe solid principles of language design. At times, these run into considerations of implementation difficulty, usability difficulties, and above all human nature that doesn’t always find blind consistency sensible and intuitive. In such cases, all languages must make judgment calls that are ultimately subjective and are about balance, flexibility, and good taste more than anything else. In my opinion, at least, D compares very favorably with other languages that inevitably have had to make similar decisions.

Intended Audience

This book assumes you’re a programmer, meaning that you know how to accomplish typical programming tasks in a language of your choice. Knowledge of any language in particular is not assumed or particularly recommended. If you know one of the Algol-derived languages (C, C++, Java, or C#), you will enjoy a slight advantage because the syntax will feel familiar from the get-go and the risk of finding false friends (similar syntax with different semantics) is minimal. (In particular, if you paste a piece of C code into a D file, it either compiles with the same semantics or doesn’t compile at all.)

A book introducing a language would be boring and incomplete without providing insight into the motivation behind various features, and without explaining the most productive ways to use those features to accomplish concrete tasks. This book discusses the rationale behind all non-obvious features and often explains why apparently better design alternatives weren’t chosen. Certain design choices may disproportionately aggravate the implementation effort, interact poorly with other features that have stronger reasons to stay put, have hidden liabilities that are invisible in short and simple examples, or simply aren’t powerful enough to pull their own weight. Above all, language designers are as fallible as any other human, so it’s very possible that good design choices exist that simply haven’t been seen.

Organization of the Book

The first chapter is a brisk walk through the major parts of the language. At that point, not all details are thoroughly explored, but you can get a good feel for the language and build expertise to write small programs in it. Chapters 2 and 3 are the obligatory reference chapters for expressions and statements, respectively. I tried to combine the re-

quired uniform thoroughness with providing highlights of the “deltas,” differences from traditional languages. With luck, you’ll find these chapters easy to read sequentially and also handy to return to for reference. The tables at the end of these chapters are “cheat sheets”—quick refreshers expressed in terse, intuitive terms.

Chapter 4 describes built-in arrays, associative arrays, and strings. Arrays can be thought of as pointers with a safety switch and are instrumental in D’s approach to memory safety and in your enjoyment of the language. Strings are arrays of UTF-encoded Unicode characters. Unicode support throughout the language and the standard library makes string handling correct and effective.

After reading the first four chapters, you can use the abstractions provided by the language to write short script-style programs. Subsequent chapters introduce abstraction building blocks. Chapter 5 describes functions in an integrated manner that includes compile-time parameterized functions (template functions) and functions evaluated during compilation. Such concepts would normally be confined to an advanced chapter, but D makes them simple enough to justify early introduction.

Chapter 6 discusses object-oriented design with classes. Again, compile-time parameterized classes are presented in an integrated, organic manner. Chapter 7 introduces additional types, notably `struct`, which is instrumental in building high-efficiency abstractions, often in concert with classes.

The following four chapters describe features that are relatively separate and specialized. Chapter 8 deals with type qualifiers. Qualifiers provide strong guarantees that are very handy in single-threaded and multithreaded applications alike. Chapter 9 covers the exception model. Chapter 10 introduces D’s powerful facilities for Contract Programming and is intentionally separate from Chapter 9 in an attempt to dispel the common misconception that error handling and Contract Programming are practically the same topic; they aren’t, and Chapter 10 explains why.

Chapter 11 gives information and advice for building large programs out of components and also gives a brief tour through D’s standard library. Chapter 12 covers operator overloading, without which a host of abstractions such as complex numbers would be severely affected. Finally, Chapter 13 discusses D’s original approach to concurrency.

A Brief History

Cheesy as it sounds, D is a work of love. Walter Bright, a C and C++ compiler writer, decided one day in the 1990s that he didn’t want to continue his career maintaining his compilers, so he set out to define a language as he thought “it should be done.” Many of us dream at some point or another of defining the Right Language; luckily, Walter already had a significant portion of the infrastructure handy—a back-end code generator, a linker, and most of all extensive experience with building language processors. The latter skill offered Walter an interesting perspective. Through some mysterious law of nature, poor language feature design reflects itself, in a Dorian Gray-esque manner, in

convoluted compiler implementation. In designing his new language, Walter attempted systematically to eliminate such disfluencies.

The then-nascent language was similar to C++ in spirit so the community called it simply D, in spite of Walter's initial attempt to dub it Mars. Let's call that language D1 for reasons that will become apparent soon. Walter worked on D1 for years and through sheer passion and perseverance amassed a growing crowd of followers. By 2006 D1 had grown into a strong language that could technically compete head to head with much more established languages such as C++ and Java. However, by that time it had become clear that D1 would not become mainstream because it did not have enough compelling features to make up for the backing that other languages had. At that time Walter decided to make a daring gambit: he decided that D1 would be the mythical throwaway first version, put D1 in maintenance mode, and embarked on a revamped design for the second iteration of the language that had the discretion to break backward compatibility. Current D1 users continued to benefit from bug fixes, but D1 would not add new features; D2 would become the flagship language definition, which I'll henceforth call D.

The gambit paid off. The first design iteration provided insights into things to do and things to avoid. Also, there was no rush to advertise the new language—newcomers could work with the stable, actively maintained D1. Since compatibility and deadline pressures were not major issues, there was time to analyze design alternatives carefully and to make the right decisions through and through. To further help the design effort, Walter also enlisted the help of collaborators such as Bartosz Milewski and me. Important features pertaining to D's approach to immutability, generic programming, concurrency, functional programming, safety, and much more were decided in long, animated meetings among the three of us at a coffee shop in Kirkland, WA.

In time, D firmly outgrew its “better C++” moniker and became a powerful multi-purpose language that could gainfully steal work from system-level, enterprise, and scripting languages alike. There was one problem left—all of this growth and innovation has happened in obscurity; little has been documented about the way D approaches programming.

The book you're now reading attempts to fill that void. I hope you will enjoy reading it as much as I enjoyed writing it.

Acknowledgments

D has a long list of contributors that I can't hope to produce in its entirety. Of these, the participants in the Usenet newsgroup `digitalmars.D` stand out. The newsgroup has acted as a sounding board for the designs we brought up for scrutiny and also generated many ideas and improvements.

Walter has benefited from community help with defining the reference implementation `dmd`, and two contributors stand out: Sean Kelly and Don Clugston. Sean has rewritten and improved the core runtime library (including the garbage collector) and

has also authored most of D's concurrency library implementation. He's very good at what he does, which sadly means that bugs in your concurrent code are more likely to be yours than his. Don is an expert in math in general and floating point numerics issues in particular. He has enormously helped D's numeric primitives to be some of the best around and has pushed D's generational abilities to their limit. As soon as the source code for the reference compiler was made available, Don couldn't resist adding to it, becoming the second-largest dmd contributor. Both Sean and Don initiated and carried through proposals that improved D's definition. Last but not least, they are awesome hackers all around and very enjoyable to interact with in person and online. I don't know where the language would be without them.

For this book, I'd like to warmly thank my reviewers for the generosity with which they carried out a difficult and thankless job. Without them this book would not be what it now is (so if you don't like it, take solace—just imagine how much worse it could have been). So allow me to extend my thanks to Alejandro Aragón, Bill Baxter, Kevin Bealer, Travis Boucher, Mike Casinghino, Álvaro Castro Castilla, Richard Chang, Don Clugston, Stephan Dilly, Karim Filali, Michel Fortin, David B. Held, Michiel Helvensteijn, Bernard Helyer, Jason House, Sam Hu, Thomas Hume, Graham St. Jack, Robert Jacques, Christian Kamm, Daniel Keep, Mark Kegel, Sean Kelly, Max Khesin, Simen Kjaeraas, Cody Koeninger, Denis Koroskin, Lars Kyllingstad, Igor Lesik, Eugene Letuchy, Pelle Måansson, Miura Masahiro, Tim Matthews, Scott Meyers, Bartosz Milewski, Fawzi Mohamed, Ellery Newcomer, Eric Niebler, Mike Parker, Derek Parnell, Jeremie Pelletier, Pablo Ripolles, Brad Roberts, Michael Rynn, Foy Savas, Christof Schardt, Steve Schveighoffer, Benjamin Shropshire, David Simcha, Tomasz Stachowiak, Robert Stewart, Knut Erik Teigen, Cris-tian Vlăsceanu, and Leor Zolman.

Andrei Alexandrescu
Sunday, May 2, 2010

This page intentionally left blank

13

Concurrency

Convergence of various factors in the hardware industry has led to qualitative changes in the way we are able to access computing resources, which in turn prompts profound changes in the ways we approach computing and in the language abstractions we use. Concurrency is now virtually everywhere, and it is software's responsibility to tap into it.

Although the software industry as a whole does not yet have ultimate responses to the challenges brought about by the concurrency revolution, D's youth allowed its creators to make informed decisions regarding concurrency without being tied down by obsoleted past choices or large legacy code bases. A major break with the mold of concurrent imperative languages is that D does not foster sharing of data between threads; by default, concurrent threads are virtually isolated by language mechanisms. Data sharing is allowed but only in limited, controlled ways that offer the compiler the ability to provide strong global guarantees.

At the same time, D remains at heart a systems programming language, so it does allow you to use a variety of low-level, maverick approaches to concurrency. (Some of these mechanisms are not, however, allowed in safe programs.)

In brief, here's how D's concurrency offering is layered:

- The flagship approach to concurrency is to use isolated threads or processes that communicate via messages. This paradigm, known as *message passing*, leads to safe and modular programs that are easy to understand and maintain. A variety of languages and libraries have used message passing successfully. Historically message passing has been slower than approaches based on memory sharing—which explains why it was not unanimously adopted—but that trend has recently undergone a definite and lasting reversal. Concurrent D programs are encouraged

to use message passing, a paradigm that benefits from extensive infrastructure support.

- D also provides support for old-style synchronization based on critical sections protected by mutexes and event variables. This approach to concurrency has recently come under heavy criticism because of its failure to scale well to today's and tomorrow's highly parallel architectures. D imposes strict control over data sharing, which in turn curbs lock-based programming styles. Such restrictions may seem quite harsh at first, but they cure lock-based code of its worst enemy: low-level data races. Data sharing remains, however, the most efficient means to pass large quantities of data across threads, so it should not be neglected.
- In the tradition of system-level languages, D programs not marked as `@safe` may use casts to obtain hot, bubbly, unchecked data sharing. The correctness of such programs becomes largely your responsibility.
- If that level of control is insufficient for you, you can use `asm` statements for ultimate control of your machine's resources. To go any lower-level than that, you'd need a miniature soldering iron and a very, very steady hand.

Before getting into the thick of these topics, let's take a brief detour in order to gain a better understanding of the hardware developments that have shaken our world.

13.1 Concurrentgate

When it comes to concurrency, we are living in the proverbial interesting times more than ever before. Interesting times come in the form of a mix of good and bad news that contributes to a complex landscape of trade-offs, forces, and trends.

The good news is that density of integration is still increasing by Moore's law; with what we know and what we can reasonably project right now, that trend will continue for at least one more decade after the time of this writing. Increased miniaturization begets increased computing power density because more transistors can be put to work together per area unit. Since components are closer together, connections are also shorter, which means faster local interconnectivity. It's an efficiency bonanza.

Unfortunately, there are a number of sentences starting with "unfortunately" that curb the enthusiasm around increased computational density. For one, connectivity is not only local—it forms a hierarchy [16]: closely connected components form units that must connect to other units, forming larger units. In turn, the larger units also connect to other larger units, forming even larger functional blocks, and so on. Connectivity-wise, such larger blocks remain "far away" from each other. Worse, increased complexity of each block increases the complexity of connectivity between blocks, which is achieved by reducing the thickness of wires and the distance between them. That means an increase of resistance, capacity, and crosstalk. Resistance and capacity worsen propagation speed in the wire. Crosstalk is the propensity of the signal in one wire to

propagate to a nearby wire by (in this case) electromagnetic field. At high frequencies, a wire is just an antenna and crosstalk becomes so unbearable that serial communication increasingly replaces parallel communication (a somewhat counterintuitive phenomenon visible at all scales—USB replaced the parallel port, SATA replaced PATA as the disk data connector, and serial buses are replacing parallel buses in memory subsystems, all because of crosstalk. Where are the days when parallel was fast and serial was slow?).

Also, the speed gap between processing elements and memory is also increasing. Whereas memory density has been increasing at predictably the same rate as general integration density, its access speed is increasingly lagging behind computation speed for a variety of physical, technological, and market-related reasons [22]. It is unclear at this time how the speed gap could be significantly reduced, and it is only growing. Hundreds of cycles may separate the processor from a word in memory; only a few years ago, you could buy “zero wait states” memory chips accessible in one clock cycle.

The existence of a spectrum of memory architectures that navigate different trade-offs among density, price, and speed, has caused an increased sophistication of memory hierarchies; accessing one memory word has become a detective investigation that involves questioning several cache levels, starting with precious on-chip static RAM and going possibly all the way to mass storage. Conversely, a given datum could be found replicated in a number of places throughout the cache hierarchy, which in turn influences programming models. We can't afford anymore to think of memory as a big, monolithic chunk comfortably shared by all processors in a system: caches foster local memory traffic and make shared data an illusion that is increasingly difficult to maintain [37].

In related, late-breaking news, the speed of light has obstinately decided to stay constant (immutable if you wish) at about 300,000,000 meters per second. The speed of light in silicon oxide (relevant to signal propagation inside today's chips) is about half that, and the speed we can achieve today for transmitting actual data is significantly below that theoretical limit. That spells more trouble for global interconnectivity at high frequencies. If we wanted to build a 10GHz chip, under ideal conditions it would take three cycles just to transport a bit across a 4.5-centimeter-wide chip while essentially performing no computation.

In brief, we are converging toward processors of very high density and huge computational power that are, however, becoming increasingly isolated and difficult to reach and use because of limits dictated by interconnectivity, signal propagation speed, and memory access speed.

The computing industry is naturally flowing around these barriers. One phenomenon has been the implosion of the size and energy required for a given computational power; today's addictive portable digital assistants could not have been fabricated at the same size and capabilities with technology only five years old. Today's trends, however, don't help traditional computers that want to achieve increased computational power at about the same size. For those, chip makers decided to give up the

battle for faster clock rates and instead decided to offer computing power packaged in already known ways: several identical central processing unit (CPUs) connected to each other and to memory via buses. Thus, in a matter of a few short years, the responsibility for making computers faster has largely shifted from the hardware crowd to the software crowd. More CPUs may seem like an advantageous proposition, but for regular desktop computer workloads it becomes tenuous to gainfully employ more than around eight processors. Future trends project an exponential expansion of the number of available CPUs well into the dozens, hundreds, and thousands. To speed up one given program, a lot of hard programming work is needed to put those CPUs to good use.

The computing industry has always had moves and shakes caused by various technological and human factors, but this time around we seem to be at the end of the rope. Since only a short time ago, taking a vacation is not an option for increasing the speed of your program. It's a scandal. It's an outrage. It's Concurrentgate.

13.2 A Brief History of Data Sharing

One aspect of the shift happening in computing is the suddenness with which processing and concurrency models are changing today, particularly in comparison and contrast to the pace of development of programming languages and paradigms. It takes years and decades for programming languages and their associated styles to become imprinted into a community's lore, whereas changes in concurrency matters turned a definite exponential elbow starting around the beginning of the 2000s.

For example, our yesteryear understanding of general concurrency¹ was centered around time sharing, which in turn originated with the mainframes of the 1960s. Back then, CPU time was so expensive, it made sense to share the CPU across multiple programs controlled from multiple consoles so as to increase overall utilization. A *process* was and is defined as the state and the resources of a running program. To implement time sharing, the CPU uses a timer interrupt in conjunction with a software scheduler. Upon each timer interrupt, the scheduler decides which process gets CPU time for the next time quantum, thus giving the illusion that several processes are running simultaneously, when in fact they all use the same CPU.

To prevent buggy processes from stomping over one another and over operating system code, *hardware memory protection* has been introduced. In today's systems, memory protection is combined with *memory virtualization* to ensure robust process isolation: each process thinks it "owns" the machine's memory, whereas in fact a translation layer from logical addresses (as the process sees memory) to physical addresses (as the machine accesses memory) intermediates all interaction of processes with memory and isolates processes from one another. The good news is that runaway processes can harm only themselves, but not other processes or the operating system kernel. The less

1. The following discussion focuses on general concurrency and does not discuss vector operation parallelization and other specialized parallel kernels.

good news is that upon each task switching, a potentially expensive swapping of address translation paraphernalia also has to occur, not to mention that every just-switched-to process wakes up with cache amnesia as the global shared cache was most likely used by other processes. And that's how *threads* were born.

A thread is a process without associated address translation information—a bare execution context: processor state plus stack. Several threads share the address space of a process, which means that threads are relatively cheap to start and switch among, and also that they can easily and cheaply share data with each other. Sharing memory across threads running against one CPU is as straightforward as possible—one thread writes, another reads. With time sharing, the order in which data is written by one thread is naturally the same as the order in which those writes are seen by others. Maintaining higher-level data invariants is ensured by using interlocking mechanisms such as critical sections protected by synchronization primitives (such as semaphores and mutexes). Through the late twentieth century, a large body of knowledge, folklore, and anecdotes has grown around what could be called “classic” multithreaded programming, characterized by shared address space, simple rules for memory effect visibility, and mutex-driven synchronization. Other models of concurrency existed, but classic multithreading was the most used on mainstream hardware.

Today’s mainstream imperative languages such as C, C++, Java, or C# have been developed during the classic multithreading age—the good old days of simple memory architectures, straightforward data sharing, and well-understood interlocking primitives. Naturally, languages modeled the realities of that hardware by accommodating threads that all share the same memory. After all, the very definition of multithreading entails that all threads share the same address space, unlike operating system processes. In addition, message-passing APIs (such as the MPI specification [29]) have been available in library form, initially for high-end hardware such as (super)computer clusters.

During the same historical period, the then-nascent functional languages adopted a principled position based on mathematical purity: we’re not interested in modeling hardware, they said, but we’d like to model math. And math for the most part does not have mutation and is time-invariant, which makes it an ideal candidate for parallelization. (Imagine the moment when those first mathematicians-turned-programmers heard about concurrency—they must have slapped their foreheads: “Wait a *minute!*...”) It was well noted in functional programming circles that such a computational model does inherently favor out-of-order, concurrent execution, but that potential was more of a latent energy than a realized goal until recent times.

Finally, Erlang was developed starting in the late 1980s as a domain-specific embedded language for telephony applications. The domain required tens of thousands of simultaneous programs running on the same machine and strongly favored a message-passing, “fire-and-forget” communication style. Although mainstream hardware and operating systems were not optimized for such workloads, Erlang initially ran on specialized hardware. The result was a language that originally combined an impure func-

tional style with heavy concurrency abilities and a staunch message-passing, no-sharing approach to communication.

Fast-forward to the 2010s. Today, even run-of-the-mill machines have more than one processor, and the decade's main challenge is to stick ever more CPUs on a chip. This has had a number of consequences, the most important being the demise of seamless shared memory.

One time-shared CPU has one memory subsystem attached to it—with buffers, several levels of caches, the works. No matter how the CPU is time-shared, reads and writes go through the same pipeline; as such, a coherent view of memory is maintained across all threads. In contrast, multiple interconnected CPUs cannot afford to share the cache subsystem: such a cache would need multiport access (expensive and poorly scalable) and would be difficult to place in the proximity of all CPUs simultaneously. Therefore, today's CPUs, almost without exception, come with their own dedicated cache memory. The hardware and protocols connecting the CPU + cache combos together are a crucial factor influencing multiprocessor system performance.

The existence of multiple caches makes data sharing across threads devilishly difficult. Now reads and writes in different threads may hit different caches, so sharing data from one thread to another is not straightforward anymore and, in fact, becomes a message passing of sorts.² for any such sharing, a sort of handshake must occur among cache subsystems to ensure that shared data makes it from the latest writer to the reader and also to the main memory.

As if things weren't interesting enough already, cache synchronization protocols add one more twist to the plot: they manipulate data in blocks, not individual word reads and word writes. This means that communicating processors "forget" the exact order in which data was written, leading to paradoxical behavior that apparently defies causality and common sense: one thread writes *x* and then *y* and for a while another thread sees the new *y* but only the old *x*. Such causality violations are extremely difficult to integrate within the general model of classic multithreading, which is imbued with the intuition of time slicing and with a simple memory model. Even the most expert programmers in classic multithreading find it unbelievably difficult to adapt their programming styles and patterns to the new memory architectures.

To illustrate the rapid changes in today's concurrency world and also the heavy influence of data sharing on languages' approach to concurrency, consider the following piece of advice given in the 2001 edition of the excellent book *Effective Java* [8, Item 51, page 204]:

When multiple threads are runnable, the thread scheduler determines which threads get to run and for how long. ... The best way to write a robust, responsive, portable multithreaded application is to ensure that there are few runnable threads at any given time.

2. This is ironic because shared memory has been faster than message passing in the classic multithreading days.

One startling detail for today’s observer is that single-processor, time-sliced threading is not only addressed by the quote above, but actually assumed without being stated. Naturally, the book’s 2008 edition³ [9] changes the advice to “ensure that the average number of runnable threads is not significantly greater than the number of processors.” Interestingly, even that advice, although it looks reasonable, makes a couple of unstated assumptions: one, that there will be high data contention between threads, which in turn causes degradation of performance due to interlocking overheads; and two, that the number of processors does not vary dramatically across machines that may execute the program. As such, the advice is contrary to that given, repeatedly and in the strongest terms, in the *Programming Erlang* book [5, Chapter 20, page 363]:

Use Lots of Processes This is important—we have to keep the CPUs busy. All the CPUs must be busy all the time. The easiest way to achieve this is to have lots of processes.⁴ When I say lots of processes, I mean lots in relation to the number of CPUs. If we have lots of processes, then we won’t need to worry about keeping the CPUs busy.

Which recommendation is correct? As usual, it all depends. The first recommendation works well on 2001-vintage hardware; the second works well in scenarios of intensive data sharing and consequently high contention; and the third works best in low-contention, high-CPU-count scenarios.

Because of the increasing difficulty of sharing memory, today’s trends make data sharing tenuous and favor functional and message-passing approaches. Not incidentally, recent years have witnessed an increased interest in Erlang and other functional languages for concurrent applications.

13.3 Look, Ma, No (Default) Sharing

In the wake of the recent hardware and software developments, D chose to make a radical departure from other imperative languages: yes, D does support threads, but they do not share any mutable data by default—they are isolated from each other. Isolation is not achieved via hardware as in the case of processes, and it is not achieved through runtime checks; it is a natural consequence of the way D’s type system is designed.

Such a decision is inspired by functional languages, which also strive to disallow all mutation and consequently mutable sharing. There are two differences. First, D programs can still use mutation freely—it’s just that mutable data is not unwittingly accessible to other threads. Second, no sharing is a *default* choice, not the *only* one. To define data as being shared across threads, you must qualify its type with `shared`. Consider, for example, two simple module-scope definitions:

3. Even the topic title was changed from “Threads” to “Concurrency” to reflect the fact that threads are but one concurrency model.

4. Erlang processes are distinct from OS processes.

```
int perThread;  
shared int perProcess;
```

In most languages, the first definition (or its syntactic equivalent) would introduce a global variable used by all threads; however, in D, `perThread` has a separate copy for each thread. The second declaration allocates only one `int` that is shared across all threads, so in a way it is closer (but not identical) to a traditional global variable.

The variable `perThread` is stored using an operating system facility known as thread-local storage (TLS). The access speed of TLS-allocated data is dependent upon the compiler implementation and the underlying operating system. Generally it is negligibly slower than accessing a regular global variable in a C program, for example. In the rare cases when that may be a concern, you may want to load the global into a stack variable in access-intensive loops.

This setup has two important advantages. First, default-share languages must carefully synchronize access around global data; that is not necessary for `perThread` because it is private to each thread. Second, the `shared` qualifier means that the type system and the human user are both in the know that `perProcess` is accessed by multiple threads simultaneously. In particular, the type system will actively guard the use of shared data and disallow uses that are obviously mistaken. This turns the traditional setup on its head: under a default-share regime, the programmer must keep track manually of which data is shared and which isn't, and indeed most concurrency-related bugs are caused by undue or unprotected sharing. Under the explicit `shared` regime, the programmer knows for sure that data *not* marked as `shared` is never indeed visible to more than one thread. (To ensure that guarantee, `shared` values undergo additional checks that we'll get to soon.)

Using `shared` data remains an advanced topic because although low-level coherence is automatically ensured by the type system, high-level invariants may not be. To provide safe, simple, and efficient communication between threads, the preferred method is to use a paradigm known as *message passing*. Memory-isolated threads communicate by sending each other asynchronous messages, which consist simply of D values packaged together.

Isolated workers communicating via simple channels are a very robust, time-proven approach to concurrency. Erlang has done that for years, as have applications based on the Message Passing Interface (MPI) specification [29].

To add acclaim to remedy,⁵ good programming practice even in default-share multi-threaded languages actually enshrines that threads ought to be isolated. Herb Sutter, a world-class expert in concurrency, writes in an article eloquently entitled “Use threads correctly = isolation + asynchronous messages” [54]:

5. That must be an antonym for the phrase “to add insult to injury.”

Threads are a low-level tool for expressing asynchronous work. “Uplevel” them by applying discipline: strive to make their data private, and have them communicate and synchronize using asynchronous messages. Each thread that needs to get information from other threads or from people should have a message queue, whether a simple FIFO queue or a priority queue, and organize its work around an event-driven message pump mainline; replacing spaghetti with event-driven logic is a great way to improve the clarity and determinism of your code.

If there is one thing that decades of computing have taught us, it must be that discipline-oriented programming does not scale. It is reassuring, then, to reckon that the quote above pretty much summarizes quite accurately the following few sections, save for the discipline part.

13.4 Starting a Thread

To start a thread, use the `spawn` function like this:

```
import std.concurrency, std.stdio;

void main() {
    auto low = 0, high = 100;
    spawn(&fun, low, high);
    foreach (i; low .. high) {
        writeln("Main thread: ", i);
    }
}

void fun(int low, int high) {
    foreach (i; low .. high) {
        writeln("Secondary thread: ", i);
    }
}
```

The `spawn` function takes the address of a function `&fun` and a number of arguments $\langle a_1 \rangle, \langle a_2 \rangle, \dots, \langle a_n \rangle$. The number of arguments n and their types must match `fun`'s signature, that is, the call `fun(\langle a_1 \rangle, \langle a_2 \rangle, \dots, \langle a_n \rangle)` must be correct. This check is done at compile time. `spawn` creates a new execution thread, which will issue the call `fun(\langle a_1 \rangle, \langle a_2 \rangle, \dots, \langle a_n \rangle)` and then terminate. Of course, `spawn` does not wait for the thread to terminate—it returns as soon as the thread is created and the arguments are passed to it (in this case, two integers).

The program above outputs a total of 200 lines to the standard output. The interleaving of lines depends on a variety of factors; it's possible that you would see 100 lines

from the main thread followed by 100 lines from the secondary thread, the exact opposite, or some seemingly random interleaving. There will never be, however, a mix of two messages on the same line. This is because `writeln` is defined to make each call atomic with regard to its output stream. Also, the order of lines emitted by each thread will be respected.

Even if the execution of `main` may end before the execution of `fun` in the secondary thread, the program patiently waits for all threads to finish before exiting. This is because the runtime support library follows a little protocol for program termination, which we'll discuss later; for now, let's just note that other threads don't suddenly die just because `main` returns.

As promised by the isolation guarantee, the newly created thread shares nothing with the caller thread. Well, almost nothing: the global file handle `stdout` is *de facto* shared across the two threads. But there is no cheating: if you look at the `std.stdio` module's implementation, you will see that `stdout` is defined as a global shared variable. Everything is properly accounted for in the type system.

13.4.1 immutable Sharing

What kind of functions can you call via `spawn`? The no-sharing stance imposes certain restrictions—you may use only by-value parameters for the thread starter function (`fun` in the example above). Any pass by reference, either explicit (by use of a `ref` parameter) or implicit (e.g., by use of an array) should be verboten. With that in mind, let's take a look at the following rewrite of the example:

```
import std.concurrency, std.stdio;

void main() {
    auto low = 0, high = 100;
    auto message = "Yeah, hi #";
    spawn(&fun, message, low, high);
    foreach (i; low .. high) {
        writeln("Main thread: ", message, i);
    }
}

void fun(string text, int low, int high) {
    foreach (i; low .. high) {
        writeln("Secondary thread: ", text, i);
    }
}
```

The rewritten example is similar to the original, but it prints an additional string. That string is created in the main thread and passed without copying into the secondary

thread. Effectively, the contents of `message` are shared between the two threads. This violates the aforementioned principle that all data sharing must be explicitly marked through the use of the `shared` keyword. Yet the example compiles and runs. What is happening?

Chapter 8 explains that `immutable` provides a strong guarantee: an `immutable` value is guaranteed never to change throughout its lifetime. The same chapter explains (§ 8.2 on page 291) that the type `string` is actually an alias for `immutable(char)[]`. Finally, we know that all contention is caused by sharing of *writable* data—as long as nobody changes it, you can share data freely as everybody will see the exact same thing. The type system and the entire threading infrastructure acknowledge that fact by allowing all `immutable` data to be freely sharable across threads. In particular, `string` values can be shared because their characters can't be changed. In fact, a large part of the motivation behind introducing `immutable` into the language was the help it brings with sharing structured data across threads.

13.5 Exchanging Messages between Threads

Threads that print messages with arbitrary interleavings are hardly interesting. Let's modify the example to ensure that threads work in tandem to print messages as follows:

```
Main thread: 0
Secondary thread: 0
Main thread: 1
Secondary thread: 1
...
Main thread: 999
Secondary thread: 999
```

To achieve that, we need to define a little protocol between the two threads: the main thread should send the message “Print this number” to the secondary thread, and the secondary thread must answer back, “Done printing.” There is hardly any concurrency going on, but the example serves well the purpose of explaining pure communication. In real applications, threads should spend most of their time doing useful work and spend relatively little time communicating with each other.

First off, in order for two threads to communicate, they need to know how to address each other. A program may have many threads chattering away, so an identification means is necessary. To address a thread, you must get a grip on its *thread id*, nicknamed henceforth as “tid,” which is returned by `spawn`. (The name of a tid's type is actually `Tid`.) In turn, the secondary thread also needs a tid to send the response back. That's easy to do by having the sender specify its own `Tid` the same way you'd write the sender's address on a snail mail envelope. Here's what the code looks like:

```
import std.concurrency, std.stdio;
```

```

void main() {
    auto low = 0, high = 100;
    auto tid = spawn(&writer);
    foreach (i; low .. high) {
        writeln("Main thread: ", i);
        tid.send(thisTid, i);
        enforce(receiveOnly!Tid() == tid);
    }
}

void writer() {
    for (;;) {
        auto msg = receiveOnly!(Tid, int())();
        writeln("Secondary thread: ", msg[1]);
        msg[0].send(thisTid);
    }
}

```

This time around `writer` takes no more arguments because it receives the information it needs in the form of messages. The main thread saves the `Tid` returned by `spawn` and then uses it in the call to the `send` method. The call sends two pieces of data to the other thread: the current thread's `Tid`, accessed via the global property `thisTid`, and the integer to be printed. After throwing that data over the fence to the other thread, the main thread waits for acknowledgment in the form of a call to `receiveOnly`. The `send` and `receiveOnly` functions work in tandem: one call to `send` in one thread is met by a call to `receiveOnly` in the other. The “only” in `receiveOnly` is present because `receiveOnly` accepts only specific types—for example, in the call `receiveOnly!bool()`, the caller accepts only a message consisting of a `bool` value; if another thread sends anything else, `receiveOnly` throws a `MessageMismatch` exception.

Let's leave `main` rummaging around the `foreach` loop and focus on `writer`'s implementation, which implements the other side of the mini-protocol. `writer` spends time in a loop starting with the receipt of a message that must consist of a `Tid` and an `int`. That's what the call `receiveOnly!(Tid, int)()` ensures; again, if the main thread sent a message with some different number or types of arguments, `receiveOnly` would fail by throwing an exception. As written, the `receiveOnly` call in `writer` matches perfectly the call `tid.send(thisTid, i)` made from `main`.

The type of `msg` is `Tuple!(Tid, int)`. Generally, messages with multiple arguments are packed in `Tuple` objects with one member per argument. If, however, the message consists only of one value, there's no redundant packing in a `Tuple`. For example, `receiveOnly!int()` returns an `int`, not a `Tuple!int`.

Continuing with `writer`, the next line performs the actual printing. Recall that for the tuple `msg`, `msg[0]` accesses the first member (i.e., the `Tid`) and `msg[1]` accesses the second member (the `int`). Finally, `writer` acknowledges that it finished writing to the console by simply sending its own `Tid` back to the sender—a sort of a blank letter that only confirms the originating address. “Yes, I got your message,” the empty letter implies, “and acted upon it. Your turn.” The main thread waits for that confirmation before continuing its work, and the loop goes on.

Sending back the `Tid` of the secondary thread is superfluous in this case; any dummy value, such as an `int` or a `bool`, would have sufficed. But in the general case there are many threads sending messages to one another, so self-identification becomes important.

13.6 Pattern Matching with receive

Most useful communication protocols are more complex than the one we defined above, and `receiveOnly` is quite limited. For example, it is quite difficult to implement with `receiveOnly` an action such as “receive an `int` or a `string`.”

A more powerful primitive is `receive`, which matches and dispatches messages based on their type. A typical call to `receive` looks like this:

```
receive(
    (string s) { writeln("Got a string with value ", s); },
    (int x) { writeln("Got an int with value ", x); }
);
```

The call above matches any of the following `send` calls:

```
send(tid, "hello");
send(tid, 5);
send(tid, 'a');
send(tid, 42u);
```

The first `send` call matches a `string` and is therefore dispatched to the first function literal in `receive`, and the other three match an `int` and are passed to the second function literal. By the way, the handler functions don’t need to be literals—some or all of them may be addresses of named functions:

```
void handleString(string s) { ... }
receive(
    &handleString,
    (int x) { writeln("Got an int with value ", x); }
);
```

Matching is not exact; instead, it follows normal overloading rules, by which `char` and `uint` are implicitly convertible to `int`. Conversely, the following calls will *not* be matched:

```
send(tid, "hello"w); // UTF-16 string (§ 4.5 on page 118)
send(tid, 5L);       // long
send(tid, 42.0);    // double
```

When `receive` sees a message of an unexpected type, it doesn't throw an exception (as `receiveOnly` does). The message-passing subsystem simply saves the non-matching messages in a queue, colloquially known as the thread's *mailbox*. `receive` waits patiently for the arrival of a message of a matching type in the mailbox. This makes `receive` and the protocols implemented on top of it more flexible, but also more susceptible to blocking and mailbox crowding. One communication misunderstanding is enough for a thread's mailbox to accumulate messages of the wrong type while `receive` is waiting for a message type that never arrives.

The `send/receive` combo handles multiple arguments easily by using `Tuple` as an intermediary. For example:

```
receive(
    (long x, double y) { ... },
    (int x) { ... }
);
```

matches the same messages as

```
receive(
    (Tuple!(long, double) tp) { ... },
    (int x) { ... }
);
```

A call like `send(tid, 5, 6.3)` matches the first function literal in both examples above.

To allow a thread to take contingency action in case messages are delayed, `receive` has a variant `receiveTimeout` that expires after a specified time. The expiration is signaled by `receiveTimeout` returning `false`:

```
auto gotMessage = receiveTimeout(
    1000, // Time in milliseconds
    (string s) { writeln("Got a string with value ", s); },
    (int x) { writeln("Got an int with value ", x); }
);
if (!gotMessage) {
    stderr.writeln("Timed out after one second.");
}
```

13.6.1 First Match

Consider the following example:

```
receive(
    (long x) { ... },
    (string x) { ... },
    (int x) { ... }
);
```

This call will not compile: `receive` rejects the call because the third handler could never be reached. Any `int` sent down the pipe stops at the first handler.

In `receive`, the order of arguments dictates how matches are attempted. This is similar, for example, to how `catch` clauses are evaluated in a `try` statement but is unlike object-oriented function dispatch. Reasonable people may disagree on the relative qualities of first match and best match; suffice it to say that first match seems to serve this particular form of `receive` quite well.

The compile-time enforcement performed by `receive` is simple: for any message types `<Msg1>` and `<Msg2>` with `<Msg2>`'s handler coming after `<Msg1>`'s in the `receive` call, `receive` makes sure that `<Msg2>` is *not* convertible to `<Msg1>`. If it is, that means `<Msg1>` will match messages of type `<Msg2>` so compilation of the call is refused. In the example above, the check fails when `<Msg1>` is `long` and `<Msg2>` is `int`.

13.6.2 Matching Any Message

What if you wanted to make sure you're looking at any and all messages in a mailbox—for example, to make sure it doesn't get filled with junk mail?

The answer is simple—just accept the type `Variant` in the last position of `receive`, like this:

```
receive(
    (long x) { ... },
    (string x) { ... },
    (double x, double y) { ... },
    ...
    (Variant any) { ... }
);
```

The `Variant` type defined in module `std.variant` is a dynamic type able to hold exactly one value of any other type. `receive` recognizes `Variant` as a generic holder for any message type, and as such a call to `receive` that has a handler for `Variant` will always return as soon as at least one message is in the queue.

Planting a `Variant` handler at the bottom of the message handling food chain is a good method to make sure that stray messages aren't left in your mailbox.

13.7 File Copying—with a Twist

Let's write a short program that copies files—a popular way to get acquainted with a language's file system interface. Ah, the joy of K&R's classic `getchar/putchar` example [34, Chapter 1, page 15]. Of course, the system-provided programs that copy files use buffered reads and writes and many other optimizations to accelerate transfer speed, so it would be difficult to write a competitive program, but concurrency may give an edge.

The usual approach to file copying goes like this:

1. Read data from the source file into a buffer.
2. If nothing was read, done.
3. Write the buffer into the target file.
4. Repeat from step 1.

Adding appropriate error handling completes a useful (if unoriginal) program. If you select a large enough buffer and both the source and destination files reside on the same disk, the performance of the algorithm is near optimal.

Nowadays a variety of physical devices count as file repositories, such as hard drives, thumb drives, optical disks, connected smart phones, and remotely connected network services. These devices have various latency and speed profiles and connect to the computer via different hardware and software interfaces. Such interfaces could and should be put to work in parallel, not one at a time as the “read buffer/write buffer” algorithm above prescribes. Ideally, both the source and the target device should be kept as busy as possible, something we could effect with two threads following the producer-consumer protocol:

1. Spawn one secondary thread that listens to messages containing memory buffers and writes them to the target file in a loop.
2. Read data from the source file in a newly allocated buffer.
3. If nothing was read, done.
4. Send a message containing the read buffer to the secondary thread.
5. Repeat from step 2.

In the new setup, one thread keeps the source busy and the other keeps the target busy. Depending on the nature of the source and target, significant acceleration could be obtained. If the device speeds are comparable and relatively slow compared to the bandwidth of the memory bus, the speed of copying could theoretically be doubled. Let's write a simple producer-consumer program that copies `stdin` to `stdout`:

```
import std.algorithm, std.concurrency, std.stdio;

void main() {
```

```
enum bufferSize = 1024 * 100;
auto tid = spawn(&fileWriter);
// Read loop
foreach (immutable(ubyte)[] buffer; stdin.byChunk(bufferSize)) {
    send(tid, buffer);
}
}

void fileWriter() {
    // Write loop
    for (;;) {
        auto buffer = receiveOnly!(immutable(ubyte)[])();
        tgt.write(buffer);
    }
}
```

The program above transfers data from the main thread to the secondary thread through immutable sharing: the messages passed have the type `immutable(ubyte)[]`, that is, arrays of immutable unsigned bytes. Those buffers are acquired in the `foreach` loop by reading input in chunks of type `immutable(ubyte)[]`, each of size `bufferSize`. At each pass through the loop, one new buffer is allocated, read into, and bound to `buffer`. The `foreach` control part does most of the hard work; all the body has to do is send off the buffer to the secondary thread. As discussed, passing data around is possible because of `immutable`; if you replaced `immutable(ubyte)[]` with `ubyte[]`, the call to `send` would not compile.

13.8 Thread Termination

There's something unusual about the examples given so far, in particular `writer` defined on page 402 and `fileWriter` defined on the facing page: both functions contain an infinite loop. In fact, a closer look at the file copy example reveals that `main` and `fileWriter` understand each other well regarding copying things around but never discuss application termination; in other words, `main` does not ever tell `fileWriter`, "We're done; let's finish and go home."

Termination of multithreaded applications has always been tricky. Threads are easy to start, but once started they are difficult to finish; the application shutdown event is asynchronous and may catch a thread in the middle of an arbitrary operation. Low-level threading APIs do offer a means to forcefully terminate threads, but invariably with the cautionary note that such a function is a blunt tool that should be replaced with a higher-level shutdown protocol.

D offers a simple and robust thread termination protocol. Each thread has an *owner* thread; by default the owner is the thread that initiated the `spawn`. You can change the

current thread's owner dynamically by calling `setOwner(tid)`. Each thread has exactly one owner but a given thread may own multiple threads.

The most important manifestation of the owner/owned relationship is that when the owner thread terminates, the calls to `receive` in the owned thread will throw the `OwnerTerminated` exception. The exception is thrown only if `receive` has no more matching messages and must wait for a new message; as long as `receive` has something to fetch from the mailbox, it will not throw. In other words, when the owner thread terminates, the owned threads' calls to `receive` (or `receiveOnly` for that matter) will throw `OwnerTerminated` if and only if they would otherwise block waiting for a new message. The ownership relation is not necessarily unidirectional. In fact, two threads may even own each other; in that case, whichever thread finishes will notify the other.

With thread ownership in mind, let's take a fresh look at the file copy program on page 406. At any given moment, there are a number of messages in flight between the main thread and the secondary thread. The faster the reads are relative to writes, the more buffers will wait in the writer thread's mailbox waiting to be processed. When `main` returns, it will cause the call to `receive` to throw an exception, but not before all of the pending messages are handled. Right after the mailbox of the writer is cleared (and the last drop of data is written to the target file), the next call to `receive` throws. The writer thread exits with the `OwnerTerminated` exception, which is recognized by the runtime system, which simply ignores it. The operating system closes `stdin` and `stdout` as it always does, and the copy operation succeeds.

It may appear there is a race between the moment the last message is sent from `main` and the moment `main` returns (causing `receive` to throw). What if the exception "makes it" before the last message—or worse, before the last few messages? In fact there is no race because causality is always respected in the posting thread: the last message is posted onto the secondary thread's queue *before* the `OwnerTerminated` exception makes its way (in fact, propagating the exception is done via the same queue as regular messages). However, a race *would* exist if `main` exits while a different, third thread is posting messages onto `fileWriter`'s queue.

A similar reasoning shows that our previous simple example that writes 200 messages in lockstep is also correct: `main` exits after mailing (in the nick of time) the last message to the secondary thread. The secondary thread first exhausts the queue and then ends with the `OwnerTerminated` exception.

If you find throwing an exception too harsh a mechanism for handling a thread's exit, you can always handle `OwnerTerminated` explicitly:

```
// Ends without an exception
void fileWriter() {
    // Write loop
    for (bool running = true; running; ) {
        receive(
            (immutable(ubyte)[] buffer) { tgt.write(buffer); },

```

```
        (OwnerTerminated) { running = false; }
    );
}
stderr.writeln("Normally terminated.");
}
```

In this case, `fileWriter` returns peacefully when `main` exits and everyone's happy. But what happens in the case when the secondary thread—the writer—throws an exception? The call to the `write` function may fail if there's a problem writing data to `tgt`. In that case, the call to `send` from the primary thread will fail by throwing an `OwnedFailed` exception, which is exactly what should happen. By the way, if an owned thread exits normally (as opposed to throwing an exception), subsequent calls to `send` to that thread also fail, just with a different exception type: `OwnedTerminated`.

The file copy program is more robust than its simplicity may suggest. However, it should be said that relying on the default termination protocol works smoothly when the relationships between threads are simple and well understood. When there are many participating threads and the ownership graph is complex, it is best to establish explicit “end-of-communication” protocols throughout. In the file copy example, a simple idea would be to send by convention a buffer of size zero to signal the writer that the reading thread has finished successfully. Then the writer acknowledges termination to the reader, which finally can exit. Such an explicit protocol scales well to cases when there are multiple threads processing the data stream between the reader and the writer.

13.9 Out-of-Band Communication

Consider that you're using the presumably smart file-copying program we just defined to copy a large file from a fast local store to a slow network drive. Midway through the copy, there's a read error—the file is corrupt. That causes `read` and subsequently `main` to throw an exception while there are many buffers in flight that haven't yet been written. More generally, we saw that if the owner terminates *normally*, any blocking call to receive from its owned threads will throw. What happens if the owner exits with an exception?

If a thread terminates by means of an exception, that indicates a serious issue that must be signaled with relative urgency to the owned threads. Indeed this is carried out via an *out-of-band* message.

Recall that `receive` cares only about matching messages and lets all others accumulate in the queue. There is one amendment to that behavior. A thread may initiate an out-of-band message by calling `prioritySend` instead of `send`. The two functions accept the same parameters but exhibit different behaviors that actually manifest themselves on the receiving side. Passing a message of type `T` with `prioritySend` causes `receive` in the receiving thread to act as follows:

- If the call to receive handles type T, then the priority message will be the next message handled, even though it arrived later than other regular (non-priority) messages. Priority messages are always pushed to the beginning of the queue, so the latest priority message sent is always the first fetched by receive (even if other priority messages are already waiting).
- If the call to receive does not handle type T (i.e., would leave the message waiting in the mailbox) and if T inherits Exception, receive throws the message directly.
- If the call to receive does not handle type T and T does not inherit Exception, receive throws an exception of type PriorityMessageException!T. That exception holds a copy of the message sent in the form of a member called message.

If a thread exits via an exception, the exception OwnerFailed propagates to all of its owned threads by means of prioritySend. In the file copy program, main throwing also causes fileWriter to throw as soon as it calls receive, and the entire process terminates by printing an error message and returning a nonzero exit code. Unlike the normal termination case, there may be buffers in flight that have been read but not yet written.

13.10 Mailbox Crowding

The producer-consumer file copy program works quite well but has an important shortcoming. Consider copying a large file between two devices of different speeds, for example, copying a legally acquired movie file from an internal drive (fast) to a network drive (possibly considerably slower). In that case, the producer (the main thread) issues buffers at considerable speed, much faster than the speed with which the consumer is able to unload them in the target file. The difference in the two speeds causes a net accumulation of buffers, which may cause the program to consume a lot of memory without achieving a boost in efficiency.

To avoid mailbox crowding, the concurrency API allows setting the maximum size of a thread's message queue, and also setting the action to take in case the maximum size has been reached. The signatures of relevance here are

```
// Inside std.concurrency
void setMaxMailboxSize(Tid tid, size_t messages,
    bool(Tid) onCrowdingDoThis);
```

The call `setMaxMailboxSize(tid, messages, onCrowdingDoThis)` directs the concurrency API to call `onCrowdingDoThis(tid)` whenever a new message is to be passed but the queue already contains `messages` entries. If `onCrowdingDoThis(tid)` returns `false` or throws an exception, the new message is ignored. Otherwise, the size of the thread's queue is checked again, and if it is less than `messages`, the new message is posted to thread `tid`. Otherwise, the entire loop is resumed.

The call occurs in the caller thread, not the callee. In other words, the thread that initiates sending a message is also responsible for taking contingency action in case the maximum mailbox size of the recipient has been reached. It seems reasonable to ask why the call should not occur in the callee; that would, however, scale the wrong way in heavily threaded programs because threads with full mailboxes may become crippled by many calls from other threads attempting to send messages.

There are a few prepackaged actions to perform when the mailbox is full: block the caller until the queue becomes smaller, throw an exception, or ignore the new message. Such predefined actions are conveniently packaged as follows:

```
// Inside std.concurrency
enum OnCrowding { block, throwException, ignore }
void setMaxMailboxSize(Tid tid, size_t messages, OnCrowding doThis);
```

In our case, it's best to simply block the reader thread once the mailbox becomes too large, which we can effect by inserting the call

```
setMaxMailboxSize(tid, 1024, OnCrowding.block);
```

right after the call to `spawn`.

The following sections describe approaches to inter-thread communication that are alternative or complementary to message passing. Message passing is the recommended method of inter-thread communication; it is easy to understand, fast, well behaved, reliable, and scalable. You should descend to lower-level communication mechanisms only in special circumstances—and don't forget, “special” is not always as special as it seems.

13.11 The shared Type Qualifier

We already got acquainted with `shared` in § 13.3 on page 397. To the type system, `shared` indicates that several threads have access to a piece of data. The compiler acknowledges that reality by restricting operations on shared data and by generating special code for the accepted operations.

The global definition

```
shared uint threadsCount;
```

introduces a value of type `shared(uint)`, which corresponds to a global unsigned `int` in a C program. Such a variable is visible to all threads in the system. The annotation helps the compiler a great deal: the language “knows” that `threadsCount` is freely accessible from multiple threads and forbids naïve access to it. For example:

```
void bumpThreadsCount() {
    ++threadsCount; // Error!
```

```
// Cannot increment a shared int!
}
```

What's happening? Down at machine level, `++threadsCount` is not an atomic operation; it's a read-modify-write operation: `threadsCount` is loaded into a register, the register value is incremented, and then `threadsCount` is written back to memory. For the whole operation to be correct, these three steps need to be performed as an indivisible unit. The correct way to increment a shared integer is to use whatever specialized atomic increment primitives the processor offers, which are portably packaged in the `std.concurrency` module:

```
import std.concurrency;
shared uint threadsCount;

void bumpThreadsCount() {
    // std.concurrency defines
    //     atomicOp(string op)(ref shared uint, int)
    atomicOp!"+="(threadsCount, 1); // Fine
}
```

Because all shared data is accounted for and protected under the aegis of the language, passing shared data via `send` and `receive` is allowed.

13.11.1 The Plot Thickens: `shared` Is Transitive

Chapter 8 explains why `const` and `immutable` must be *transitive* (aka deep or recursive): following any indirections starting from an `immutable` object must keep data `immutable`. Otherwise, the `immutable` guarantee has the power of a comment in the code. You can't say something is `immutable` "up to a point" after which it changes its mind. You can, however, say that data is *mutable* up to a point, where it becomes `immutable` through and through. Stepping into immutability is veering down a one-way street. We've seen that `immutable` facilitates a number of correct and pain-free idioms, including functional style and sharing of data across threads. If immutability applied "up to a point," then so would program correctness.

The same exact reasoning goes for `shared`. In fact, with `shared` the necessity of transitivity becomes painfully obvious. Consider:

```
shared int* pInt;
```

which according to the qualifier syntax (§ 8.2 on page 291) is equivalent to

```
shared(int*) pInt;
```

The correct meaning of `pInt` is “The pointer is shared and the data pointed to by the pointer is also shared.” A shallow, non-transitive approach to sharing would make `pInt` “a shared pointer to non-shared memory,” which would be great if it weren’t untenable. It’s like saying, “I’ll share this wallet with everyone; just please remember that the money in it ain’t shared.”⁶ Claiming the pointer is shared across threads but the pointed-to data is not takes us back to the wonderful programming-by-honor-system paradigm that has failed so successfully throughout history. It’s not the voluntary malicious uses, it’s the honest mistakes that form the bulk of problems. Software is large, complex, and ever-changing, traits that never go well with maintaining guarantees through convention.

There is, however, a notion of “unshared pointer to shared data” that does hold water. Some thread holds a private pointer, and the pointer “looks” at shared data. That is easily expressible syntactically as

```
shared(int)* pInt;
```

As an aside, if there exists a “Best Form-Follows-Function” award, then the notation `qualifier(type)` should snatch it. It’s perfect. You can’t even syntactically create the wrong pointer type, because it would look like this:

```
int shared(*) pInt;
```

which does not make sense even syntactically because `(*)` is not a type (granted, it *is* a nice emoticon for a cyclops).

Transitivity of shared applies not only to pointers, but also to fields of `struct` and `class` objects: fields of a shared object are automatically qualified as shared as well. We’ll discuss in detail the ways in which `shared` interacts with `classes` and `structs` later in this chapter.

13.12 Operations with shared Data and Their Effects

Working with shared data is peculiar because multiple threads may read and write it at any moment. Therefore, the compiler makes sure that all operations preserve integrity of data and also causality of operations.

Reads and writes of shared values are allowed and guaranteed to be atomic: numeric types (save for `real`), pointers, arrays, function pointers, delegates, and class references. `struct` types containing exactly one of the mentioned types are also readable and writable atomically. Notably absent is `real`, which is the only platform-dependent type with which the implementation has discretion regarding atomic sharing. On Intel machines, `real` has 80 bits, which makes it difficult to assign atomically in 32-bit programs. Anyway, `real` is meant mostly for high-precision temporary results and not for data interchange, so it makes little sense to want to share it anyway.

6. Incidentally, you can share a wallet with theft-protected money with the help of `const` by using the type `shared(const(Money)*).`

For all numeric types and function pointers, `shared`-qualified values are convertible implicitly to and from unqualified values. Pointer conversions between `shared(T*)` and `shared(T)*` are allowed in both directions. Primitives in `std.concurrency` allow you to do arithmetic on shared numeric types.

13.12.1 Sequential Consistency of `shared` Data

With regard to the visibility of shared data operations across threads, D makes two guarantees:

- The order of reads and writes of shared data issued by one thread is the same as the order specified by the source code.
- The global order of reads and writes of shared data is some interleaving of reads and writes from multiple threads.

That seems to be a very reasonable set of assumptions—self-evident even. In fact, the two guarantees fit time-sliced threads implemented on a uniprocessor system quite well.

On multiprocessors, however, these guarantees are very restrictive. The problem is that in order to ensure the guarantees, all writes must be instantly visible throughout all threads. To effect that, shared accesses must be surrounded by special machine code instructions called *memory barriers*, ensuring that the order of reads and writes of shared data is the same as seen by all running threads. Such serialization is considerably more expensive in the presence of elaborate cache hierarchies. Also, staunch adherence to sequential consistency prevents reordering of operations, an important source of compiler-level optimizations. Combined, the two restrictions lead to dramatic slowdown—as much as one order of magnitude.

The good news is that such a speed loss occurs only with shared data, which tends to be rare. In real programs, most data is not shared and therefore need not meet sequential consistency requirements. The compiler optimizes code using non-shared data to the maximum, in full confidence that no other thread can ever access it, and only tiptoes around shared data. A common and recommended programming style with shared data is to copy shared values into thread-local working copies, work on the copies, and then write the copies back into the shared values.

13.13 Lock-Based Synchronization with `synchronized` classes

A historically popular method of writing multithreaded programs is *lock-based synchronization*. Under that discipline, access to shared data is protected by *mutexes*—synchronization objects that serialize execution of portions of the code that temporarily

break data coherence, or that might see such a temporary breakage. Such portions of code are called *critical sections*.⁷

A lock-based program's correctness is ensured by introducing ordered, serial access to shared data. A thread that needs access to a piece of shared data must acquire (lock) a mutex, operate on the data, and then release (unlock) that mutex. Only one thread at a time may acquire a given mutex, which is how serialization is effected: when several threads want to acquire the same mutex, one “wins” and the others wait nicely in line. (The way the line is served—that is, thread priority—is important and may affect applications and the operating system quite visibly.)

Arguably the “Hello, world!” of multithreaded programs is the bank account example—an object accessible from multiple threads that must expose a safe interface for depositing and withdrawing funds. The single-threaded baseline version looks like this:

```
import std.contracts;

// Single-threaded bank account
class BankAccount {
    private double _balance;
    void deposit(double amount) {
        _balance += amount;
    }
    void withdraw(double amount) {
        enforce(_balance >= amount);
        _balance -= amount;
    }
    @property double balance() {
        return _balance;
    }
}
```

In a free-threaded world, `+=` and `-=` are a tad misleading because they “look” atomic but are not—both are read-modify-write operations. Really `_balance += amount` is encoded as `_balance = _balance + amount`, which means the processor loads `_balance` and `_amount` into its own operating memory (registers or an internal stack), adds them, and deposits the result back into `_balance`.

Unprotected concurrent read-modify-write operations lead to incorrect behavior. Say your account has `_balance == 100.0` and one thread triggered by a check deposit calls `deposit(50)`. The call gets interrupted, right after having loaded `100.0` from mem-

7. A potential source of confusion is that Windows uses the term *critical section* for lightweight mutex objects that protect a critical section and *mutex* for heavier-weight mutexes that help inter-process communication.

ory, by another thread calling `withdraw(2.5)`. (That's you at the corner coffee shop getting a latte with your debit card.) Let's say the coffee shop thread finishes the entire call uninterrupted and updates `_balance` to 97.5, but that event happens unbeknownst to the deposit thread, which has loaded 100 into a CPU register already and still thinks that's the right amount. The call `deposit(50)` computes a new balance of 150 and writes that number back into `_balance`. That is a typical *race condition*. Congratulations—free coffee for you (be warned, though; buggy book examples may be rigged in your favor, but buggy production code isn't). To introduce proper synchronization, many languages offer a `Mutex` type that lock-based threaded programs use to protect access to `balance`:

```
// This is not D code
// Multithreaded bank account in a language with explicit mutexes
class BankAccount {
    private double _balance;
    private Mutex _guard;
    void deposit(double amount) {
        _guard.lock();
        _balance += amount;
        _guard.unlock();
    }
    void withdraw(double amount) {
        _guard.lock();
        try {
            enforce(_balance >= amount);
            _balance -= amount;
        } finally {
            _guard.unlock();
        }
    }
    @property double balance() {
        _guard.lock();
        double result = _balance;
        _guard.unlock();
        return result;
    }
}
```

All operations on `_balance` are now protected by acquiring `_guard`. It may seem there is no need to protect `balance` with `_guard` because a `double` can be read atomically, but protection must be there for reasons hiding themselves under multiple layers of Maya veils. In brief, because of today's aggressive optimizing compilers and relaxed memory models, *all* access to shared data must entail some odd secret handshake that

has the writing thread, the reading thread, and the optimizing compiler as participants; absolutely any bald read of shared data throws you into a world of pain (so it's great that D disallows such baldness by design). First and most obvious, the optimizing compiler, seeing no attempt at synchronization on your part, feels entitled to optimize access to `_balance` by holding it in a processor register. Second, in all but the most trivial examples, the compiler *and* the CPU feel entitled to freely reorder bald, unqualified access to shared data because they consider themselves to be dealing with thread-local data. (Why? Because that's most often the case and yields the fastest code, and besides, why hurt the plebes instead of the few and the virtuous?) This is one of the ways in which modern multithreading defies intuition and confuses programmers versed in classic multithreading. In brief, the `balance` property must be synchronized to make sure the secret handshake takes place.

To guarantee proper unlocking of `Mutex` in the presence of exceptions and early returns, languages with scoped object lifetime and destructors define an ancillary `Lock` type to acquire the lock in its constructor and release it in the destructor. The ensuing idiom is known as *scoped locking* [50] and its application to `BankAccount` looks like this:

```
// C++ version of an interlocked bank account using scoped locking
class BankAccount {
private:
    double _balance;
    Mutex _guard;
public:
    void deposit(double amount) {
        auto lock = Lock(_guard);
        _balance += amount;
    }
    void withdraw(double amount) {
        auto lock = Lock(_guard);
        enforce(_balance >= amount);
        _balance -= amount;
    }
    double balance() {
        auto lock = Lock(_guard);
        return _balance;
    }
}
```

`Lock` simplifies code and improves its correctness by automating the pairing of locking and unlocking. Java, C#, and other languages simplify matters further by embedding `_guard` as a hidden member and hoisting locking logic up to the signature of the method. In Java, the example would look like this:

```
// Java version of an interlocked bank account using
// automated scoped locking with the synchronized statement
class BankAccount {
    private double _balance;
    public synchronized void deposit(double amount) {
        _balance += amount;
    }
    public synchronized void withdraw(double amount) {
        enforce(_balance >= amount);
        _balance -= amount;
    }
    public synchronized double balance() {
        return _balance;
    }
}
```

The corresponding C# code looks similar, though `synchronized` should be replaced with `[MethodImpl(MethodImplOptions.Synchronized)]`.

Well, you've just seen the good news: in the small, lock-based programming is easy to understand and seems to work well. The bad news is that in the large, it is very difficult to pair locks with data appropriately, choose locking scope and granularity, and use locks consistently across several objects (not paying attention to the latter issue leads to threads waiting for each other in a *deadlock*). Such issues made lock-based coding difficult enough in the good ole days of classic multithreading; modern multithreading (with massive concurrency, relaxed memory models, and expensive data sharing) has put lock-based programming under increasing attack [53]. Nevertheless, lock-based synchronization is still useful in a variety of designs.

D offers limited mechanisms for lock-based synchronization. The limits are deliberate and have the advantage of ensuring strong guarantees. In the particular case of `BankAccount`, the D version is very simple:

```
// D interlocked bank account using a synchronized class
synchronized class BankAccount {
    private double _balance;
    void deposit(double amount) {
        _balance += amount;
    }
    void withdraw(double amount) {
        enforce(_balance >= amount);
        _balance -= amount;
    }
    double balance() {
```

```
    return _balance;
}
}
```

D hoists synchronized one level up to the entire class. This allows D's BankAccount to provide stronger guarantees: even if you wanted to make a mistake, there is no way to offer back-door unsynchronized access to `_balance`. If D allowed mixing synchronized and unsynchronized methods in the same class, all bets would be off. In fact, experience with method-level synchronized has shown that it's best to either define all or none as synchronized; dual-purpose classes are more trouble than they're worth.

The synchronized class-level attribute affects objects of type `shared(BankAccount)` and automatically serializes calls to any method of the class. Also, protection checks get stricter for synchronized classes. Recall that according to § 11.1 on page 337, normal protection checks ordinarily do allow access to non-public members for all code within a module. Not so for synchronized classes, which obey the following rules:

- No public data is allowed at all.
- Access to protected members is restricted to methods of the class and its descendants.
- Access to private members is restricted to methods of the class.

13.14 Field Typing in synchronized classes

The transitivity rule for shared objects dictates that a shared class object propagates the shared qualifier down to its fields. Clearly synchronized brings some additional law and order to the table, which is reflected in relaxed typechecking of fields inside the methods of synchronized classes. In order to provide strong guarantees, synchronized affects semantic checking of fields in a slightly peculiar manner, which tracks the correspondingly peculiar semantics of synchronized.

Synchronized methods' protection against races is *temporary* and *local*. The temporary aspect is caused by the fact that as soon as the method returns, fields are not protected against races anymore. The local aspect concerns the fact that synchronized ensures protection of data directly embedded inside the object, but not data indirectly referred by the object (i.e., through class references, pointers, or arrays). Let's look at each in turn.

13.14.1 Temporary Protection == No Escape

Maybe not very intuitively, the temporary nature of synchronized entails the rule that no address of a field can escape a synchronized address. If that happened, some other

portion of the code could access some data beyond the temporary protection conferred by method-level synchronization.

The compiler will reject any attempt to return a `ref` or a pointer to a field out of a method, or to pass a field by `ref` or by pointer to some function. To illustrate why that rule is sensible, consider the following example:

```
double * nyukNyuk; // N.B.: not shared

void sneaky(ref double r) { nyukNyuk = &r; }

synchronized class BankAccount {
    private double _balance;
    void fun() {
        nyukNyuk = &_balance; // Error! (as there should be)
        sneaky(_balance);    // Error! (as there should be)
    }
}
```

The first line of `fun` attempts to take the address of `_balance` and assign it to a global. If that operation were to succeed, the type system's guarantee would have failed—henceforth, the program would have shared access to data through a non-shared value. The assignment fails to typecheck. The second operation is a tad more subtle in that it attempts to do the aliasing via a function call that takes a `ref` parameter. That also fails; practically, passing a value by means of `ref` entails taking the address prior to the call. Taking the address is forbidden, so the call fails.

13.14.2 Local Protection == Tail Sharing

The protection offered by `synchronized` is also local in the sense that it doesn't necessarily protect data beyond the direct fields of the object. As soon as indirection enters into play, the guarantee that only one thread has access to data is lost. If you think of data as consisting of a “head” (the part sitting in the physical memory occupied by the `BankAccount` object) and possibly a “tail” (memory accessed indirectly), then a `synchronized` class is able to protect the “head” of the data, whereas the “tail” remains shared. In light of that reality, typing of fields of a `synchronized` class inside a method goes as follows:

- All numeric types are `not shared` (they have no tail) so they can be manipulated normally.
- Array fields declared with type `T[]` receive type `shared(T)[]`; that is, the head (the slice limits) is `not shared` and the tail (the contents of the array) remains `shared`.
- Pointer fields declared with type `T*` receive type `shared(T)*`; that is, the head (the pointer itself) is `not shared` and the tail (the pointed-to data) remains `shared`.

- Class fields declared with type T receive type shared(T). Classes are automatically by-reference, so they're "all tail."

These rules apply on top of the no-escape rule described in the previous section. One direct consequence is that operations affecting direct fields of the object can be freely reordered and optimized inside the method, as if sharing has been temporarily suspended for them—which is exactly what synchronized does.

There are cases in which an object completely owns another. Consider, for example, that the BankAccount stores all of its past transactions in a list of double:

```
// Not synchronized and generally thread-agnostic
class List(T) {
    ...
    void append(T value) {
        ...
    }
}

// Keeps a List of transactions
synchronized class BankAccount {
    private double _balance;
    private List!double _transactions;
    void deposit(double amount) {
        _balance += amount;
        _transactions.append(amount);
    }
    void withdraw(double amount) {
        enforce(_balance >= amount);
        _balance -= amount;
        _transactions.append(-amount);
    }
    double balance() {
        return _balance;
    }
}
```

The List class was not designed to be shared across threads so it does not use any synchronization mechanism, but it is in fact never shared! All of its uses are entirely private to the BankAccount object and completely protected inside synchronized methods. Assuming List does not do senseless shenanigans such as saving some internal pointer into a global variable, the code should be good to go.

Unfortunately, it isn't. Code like the above would not work in D because append is not callable against a shared(List!double) object. One obvious reason for the compiler's

refusal is that the honor system doesn't go well with compilers. `List` may be a well-behaved class and all, but the compiler would have to have somewhat harder evidence to know that there is no sneaky aliasing of shared data afoot. The compiler could, in theory, go ahead and inspect `List`'s class definition, but in turn, `List` may be using some other components found in other modules, and before you can say "interprocedural analysis," things are getting out of hand.

Interprocedural analysis is a technique used by compilers and program analyzers to prove facts about a program by looking at more functions at once. Such analyses are typically slow, scale poorly with program size, and are sworn enemies of separate compilation. Although there exist systems that use interprocedural analysis, most of today's languages (including D) do all of their typechecking without requiring it.

An alternative solution to the owned subobject problem is to add new qualifiers that describe ownership relationships such as "BankAccount owns its `_transactions` member and therefore its mutex also serializes operations on `_transactions`." With the proper annotations in place, the compiler could verify that `_transactions` is entirely encapsulated inside `BankAccount` and therefore can be safely used without worrying about undue sharing. Systems and languages that do that have been proposed [25, 2, 11, 6] but for the time being they are not mainstream. Such ownership systems introduce significant complications in the language and its compiler. With lock-based synchronization as a whole coming under attack, D shunned beefing up support for an ailing programming technique. It is not impossible that the issue might be revisited later (ownership systems have been proposed for D [42]), but for the time being certain lock-based designs must step outside the confines of the type system, as discussed next.

13.14.3 Forcing Identical Mutexes

D allows dynamically what the type system is unable to guarantee statically: an owner-owned relationship in terms of locking. The following global primitive function is accessible:

```
// Inside object.d
setSameMutex(shared Object ownee, shared Object owner);
```

A class object `obj` may call `obj.setMutex(owner)` to effectively throw away its associated synchronization object and start using the same synchronization object as `owner`. That way you can be sure that locking `owner` really locks `obj`, too. Let's see how that would work with the `BankAccount` and the `List`.

```
// Thread-aware
synchronized class List(T) {
    ...
    void append(T value) {
```

```
    ...
}

// Keeps a List of transactions
synchronized class BankAccount {
    private double _balance;
    private List!double _transactions;

    this() {
        // The account owns the list
        setSameMutex(_transactions, this);
    }
    ...
}
```

The way the scheme works requires that `List` (the owned object) be `synchronized`. Subsequent operations on `_transactions` would lock the `_transactions` field per the normal rules, but in fact they go ahead and acquire `BankAccount` object's mutex directly. That way the compiler is happy because it thinks every object is locked in separation. Also, the program is happy because in fact only one mutex controls the `BankAccount` and also the `List` subobject. Acquiring the mutex of `_transactions` is in reality acquiring the already locked mutex of `this`. Fortunately, such a recursive acquisition of an already owned, uncontested lock is relatively cheap, so the code is correct and not too locking-intensive.

13.14.4 The Unthinkable: casting Away shared

Continuing the preceding example, if you are absolutely positive that the `_transactions` list is completely private to the `BankAccount` object, you can cast away `shared` and use it without any regard to threads like this:

```
// Not synchronized and generally thread-agnostic
class List(T) {
    ...
    void append(T value) {
        ...
    }
}

synchronized class BankAccount {
    private double _balance;
    private List!double _transactions;
```

```

void deposit(double amount) {
    _balance += amount;
    (cast(List!double) _transactions).append(amount);
}
void withdraw(double amount) {
    enforce(_balance >= amount);
    _balance -= amount;
    (cast(List!double) _transactions).append(-amount);
}
double balance() {
    return _balance;
}
}

```

Now the code does compile and run. The only caveat is that now correctness of the lock-based discipline in the program is ensured by you, not by the language's type system, so you're not much better off than with languages that use default sharing. The advantage you are still enjoying is that casts are localized and can be searched for and carefully reviewed.

13.15 Deadlocks and the synchronized Statement

If the bank account example is the “Hello, world!” of threaded programs, the bank account transfer example must be the corresponding (if grimmer) introduction to threads that deadlock. The example goes like this: Assume you have two BankAccount objects, say, checking and savings. The challenge is to define an atomic transfer of some money from one account to another.

The naïve approach goes like this:

```

// Transfer version 1: non-atomic
void transfer(shared BankAccount source, shared BankAccount target,
    double amount) {
    source.withdraw(amount);
    target.deposit(amount);
}

```

This version is not atomic, however; between the two calls there is a quantum of time when money is missing from both accounts. If just at that time a thread executes the inspectForAuditing function, things may get a little tense.

To make the transfer atomic, you need to acquire the hidden mutexes of the two objects outside their methods, at the beginning of transfer. You can effect that with the help of synchronized statements:

```
// Transfer version 2: PROBLEMATIC
void transfer(shared BankAccount source, shared BankAccount target,
    double amount) {
    synchronized (source) {
        synchronized (target) {
            source.withdraw(amount);
            target.deposit(amount);
        }
    }
}
```

The synchronized statement acquires an object's hidden mutex through the execution of the statement's body. Any method call against that object benefits from an already acquired lock.

The problem with the second version of transfer is that it's prone to deadlock: if two threads attempt to execute a transfer between the same accounts but *in opposite directions*, the threads may block forever. A thread attempting to transfer money from checking to savings locks checking exactly as another thread attempting to transfer money from savings to checking manages to lock savings. At that point, each thread holds a lock, and each thread needs the other thread's lock. They will never work out an understanding.

To really fix the problem, you need to use synchronized with *two* arguments:

```
// Transfer version 3: correct
void transfer(shared BankAccount source, shared BankAccount target,
    double amount) {
    synchronized (source, target) {
        source.withdraw(amount);
        target.deposit(amount);
    }
}
```

Synchronizing on several objects in the same synchronized statement is different from successively synchronizing on each. The generated code always acquires mutexes in the same order in all threads, regardless of the syntactic order in which you specify the objects. That way, deadlock is averted.

The actual order in the reference implementation is the increasing order of object addresses. Any global ordering would work just as well.

Multi-argument synchronized is helpful but, unfortunately, not a panacea. General deadlock may occur non-locally—one mutex is acquired in one function, then another in a different function, and so on, until a deadlock cycle closes. But synchronized with

multiple arguments raises awareness of the issue and fosters correct code with modular mutex acquisition.

13.16 Lock-Free Coding with shared classes

The theory of lock-based synchronization was established in the 1960s. As early as 1972 [23], researchers started making inroads toward avoiding the slow, ham-fisted mutexes as much as possible in multithreaded programs. For example, some types were assignable atomically so people reckoned there was no ostensible need to guard such assignments with mutex acquisition. Also, some processors offered more advanced lightweight interlocked instructions such as atomic increment or test-and-set. About three decades later, in 1990, there was a definite beam of hope that some clever combination of atomic read-write registers could help avoid the tyranny of locks. At that point, a seminal piece of work had the last word in a line of work and the first word in another.

Herlihy's 1991 paper "Wait-free synchronization" [31] marked an absolutely powerful development in concurrent programming. Prior to that, it was unclear to hardware and software developers alike what kind of synchronization primitives would be best to work with. For example, a processor with atomic reads and writes for ints could intuitively be considered less powerful than one that also offers atomic `+=`. It may appear that one that offers atomic `*=` is even better; generally, the more atomic primitives one has at one's disposal, the merrier.

Herlihy blew that theory out of the water and in particular has shown that certain seemingly powerful synchronization primitives, such as test-and-set, fetch-and-add, and even one global shared FIFO queue, are virtually useless. These *impossibility results* were proven clearly enough to instantly disabuse anyone of the illusion that such mechanisms could provide the magic concurrency potion. Fortunately, Herlihy has also proved *universality results*—certain synchronization primitives may theoretically synchronize an infinite number of concurrent threads. Remarkably, the "good" primitives are not more difficult to implement than the "bad" ones and don't look particularly powerful to the naked eye. Of the useful synchronization primitives, one known as *compare-and-swap* has caught on and is implemented today by virtually all processors. Compare-and-swap has the following semantics:

```
// This function executes atomically
bool cas(T)(shared(T) * here, shared(T) ifThis, shared(T) writeThis) {
    if (*here == ifThis) {
        *here = writeThis;
        return true;
    }
    return false;
}
```

In plain language, `cas` atomically compares a memory location with a given value, and if the location is equal to that value, it stores a new value; otherwise, it does nothing. The result of the operation tells whether the store took place. The entire `cas` operation is atomic and must be provided as a primitive. The set of possible `Ts` is limited to integers of the native word size of the host machine (i.e., 32 or 64 bits). An increasing number of machines offer *double-word compare-and-swap*, sometimes dubbed `cas2`. That operation atomically manipulates 64-bit data on a 32-bit machine and 128-bit data on a 64-bit machine. In view of the increasing support for `cas2` on contemporary machines, D offers double-word compare-and-swap under the same name (`cas`) as an overloaded intrinsic function. So in D you can `cas` values of types `int`, `long`, `float`, `double`, all arrays, all pointers, and all class references.

13.16.1 shared classes

Following Herlihy's universality proofs, many data structures and algorithms took off around the nascent “`cas`-based programming.” Now, if a `cas`-based implementation is possible for theoretically any synchronization problem, nobody has said it's easy. Defining `cas`-based data structures and algorithms, and particularly proving that they work correctly, is a difficult feat. Fortunately, once such an entity is defined and encapsulated, it can be reused to the benefit of many [57].

To tap into `cas`-based lock-free goodness, use the `shared` attribute with a `class` or `struct` definition:

```
shared struct LockFreeStruct {  
    ...  
}  
  
shared class LockFreeClass {  
    ...  
}
```

The usual transitivity rules apply: `shared` propagates to the fields of the `struct` or `class`, and methods offer no special protection. All you can count on are atomic assignments, `cas` calls, the guarantee that the compiler and machine won't do any reordering of operations, and your unbridled confidence. But be warned—if coding were walking and message passing were jogging, lock-free programming would be no less than the Olympics.

13.16.2 A Couple of Lock-Free Structures

As a warmup exercise, let's implement a lock-free stack type. The basic idea is simple: the stack is maintained as a singly linked list, and insertions as well as removals proceed at the front of the list:

```
shared struct Stack(T) {
    private shared struct Node {
        T _payload;
        Node * _next;
    }
    private Node * _root;

    void push(T value) {
        auto n = new Node(value);
        shared(Node)* oldRoot;
        do {
            oldRoot = _root;
            n._next = oldRoot;
        } while (!cas(&_root, oldRoot, n));
    }

    shared(T)* pop() {
        typeof(return) result;
        shared(Node)* oldRoot;
        do {
            oldRoot = _root;
            if (!oldRoot) return null;
            result = & oldRoot->_payload;
        } while (!cas(&_root, oldRoot, oldRoot->_next));
        return result;
    }
}
```

Stack is a `shared struct`, and as a direct consequence pretty much everything inside of it is also `shared`. The internal type `Node` has the classic payload-and-pointer structure, and the `Stack` itself stores the root of the list.

The `do/while` loops in the two primitives may look a bit odd, but they are very common; slowly but surely, they dig a deep groove in the cortex of every cas-based programming expert-to-be. The way `push` works is to first create a new `Node` that will store the new value. Then, in a loop, `_root` is assigned the pointer to the new node, but *only* if in the meantime no other thread has changed it! It's quite possible that another thread has also performed a stack operation, so `push` needs to make sure that the root assumed in `oldRoot` has not changed while the new node was being primed.

The `pop` method does not return by value, but instead by pointer. This is because `pop` may find the queue empty, which is not an exceptional condition (as it would be in a single-threaded stack). For a shared stack, checking for an element, removing it, and

returning it are one organic operation. Aside from the return aspect, `pop` is similar in the implementation to `push`: `_root` is replaced with care such that no other thread changes it while the payload is being fetched. At the end of the loop, the extracted value is off the stack and can be safely returned to its caller.

If `Stack` didn't seem that complicated, let's look at actually exposing a richer singly linked interface; after all, most of the infrastructure is built inside `Stack` already.

Unfortunately, for a list things are bound to become more difficult. How much more difficult? Brutally more difficult. One fundamental problem is insertion and deletion of nodes at arbitrary positions in the list. Say we have a list of `int` containing a node with payload 5 followed by a node with payload 10, and we want to remove the 5 node. No problem here—just do the `cas` magic to swing `_root` to point to the 10 node. The problem is, if at the same time another thread inserts a new node right after the 5 node, that node will be irretrievably lost: `_root` knows nothing about it.

Several solutions exist in the literature; none of them is trivially simple. The implementation described below, first proposed by Harris [30] in the suggestively entitled paper “A pragmatic implementation of non-blocking linked-lists,” has a hackish flavor to it because it relies on setting the unused least significant bit of the `_next` pointer. The idea is first to mark that pointer as “logically deleted” by setting its bit to zero, and then to excise the node entirely in a second step:

```
shared struct SharedList(T) {
    shared struct Node {
        private T _payload;
        private Node * _next;

        @property shared(Node)* next() {
            return clearlsb(_next);
        }

        bool removeAfter() {
            shared(Node)* thisNext, afterNext;
            // Step 1: set the lsb of _next for the node to delete
            do {
                thisNext = next;
                if (!thisNext) return false;
                afterNext = thisNext._next;
            } while (!cas(&thisNext._next, afterNext, setlsb(afterNext)));
            // Step 2: excise the node to delete
            if (!cas(&_next, thisNext, afterNext)) {
                afterNext = thisNext._next;
                while (!haslsb(afterNext)) {
                    thisNext._next = thisNext._next.next;
                }
            }
        }
    };
}
```

```

        }
        _next = afterNext;
    }
}

void insertAfter(T value) {
    auto newNode = new Node(value);
    for (;;) {
        // Attempt to find an insertion point
        auto n = _next;
        while (n && haslsb(n)) {
            n = n._next;
        }
        // Found a possible insertion point, attempt insert
        auto afterN = n._next;
        newNode._next = afterN;
        if (cas(&n._next, afterN, newNode)) {
            break;
        }
    }
}

private Node * _root;

void pushFront(T value) {
    ...
}
shared(T)* popFront() {
    ...
}
}

```

The implementation is tricky but can be understood if you keep in mind a couple of invariants. First, it's OK for logically deleted nodes (i.e., `Node` objects with the field `_next` having its least significant bit set) to hang around for a little bit. Second, a node is never inserted after a logically deleted node. That way, the list stays coherent even though nodes may appear and disappear at any time.

The implementation of `clearlsb`, `setlsb` and `haslsb` is as barbaric as it gets; for example:

```
T* setlsb(T)(T* p) {
    return cast(T*)(cast(size_t)p | 1);
}
```

13.17 Summary

The implementation of `setlsb`, dirty and leaking some grease at the seams, is a fitting finale for a chapter that has started with the simple beauty of message passing and has gradually descended into the underworld of sharing.

D has an ample offering of threading amenities. For most applications on modern machines, the preferred mechanism is defining protocols built around message passing. Immutable sharing should be of great help there. You'd be well advised to use message passing for defining robust, scalable concurrent applications.

If you need to do synchronization based on mutual exclusion, you can do so with the help of `synchronized` classes. Be warned that support for lock-based programming is limited compared to other languages, and for good reasons.

If you need simple sharing of data, you may want to use `shared` values. D guarantees that operations on shared values are performed in the order specified in your code and do not cause visibility paradoxes and low-level races.

Finally, if activities such as bungee jumping, crocodile taming, or walking on coals seem sheer boredom to you, you'll be glad that lock-free programming exists, and that you can do it in D by using `shared` structs and `classes`.

Index

Non-alphabetical

!, 53
!in, 56
!is, 57
\$, 11, 31, 95, 97, 379
&, 52, 58, 124, 371
&&, 59
&=, 60
*, 52, 54
*=, 60
*, 124, 367, 371
+, 53, 55
++, 50, 53, 367, 412
+=, 60, 415
+, 367, 371
. , 60
-, 53, 55
--, 50, 53, 367
--main (compiler switch), 133
-0 (minus zero), 58
-=, 60, 415
-I (compiler switch), 340
-J (compiler switch), 37

`~`, 22, 53, 55, 60, 100, 111, 121, 367
`-=`, 103, 104
1970s, 131

A

abstract character, 118
abstract class, 218, 219
abstract, 24, 218–221
abstraction
 low-level versus high-level, 126
 mechanism, 240
access protection, 203
access specifier, 199, 261
 for **structs**, 261
accumulation, 410
accumulation function, 16, 157
adage, 337
adding method at runtime, 386, 387
addition, 55
additive expression, 55
address translation, 394, 395
address-of operator, 52, 124
aggregate contract, 331
aggregate function, 158
algebra, 165, 366, 373
algebraic type, 272
Algol, 6
algorithmic efficiency, 142
alias equality, 57
alias parameter, 148
`alias this`, 230, 263
`alias`, 149, 152, 240, 276–278, 347
aliasing, 121, 239
`align`, 268–269
alignment, 267
`alignof`, 269
allocation, 184
 cost, 180
ambiguity, 80, 81, 147, 160
ambiguous-gender type, 15
ambush, 140
ancestor class, 191

angle brackets, 11
anonymous
 unions and **structs**, 272
 class, 226
 function, *see* function literal
ANSI C, 315
antisymmetry, 144, 210
API, 329, 351
appending to array, 103
application logic, 328, 329
application shutdown, 407
approximate database search, 82
Aragón, Alejandro, xxvii
arbitrary behavior, 95
arithmetic operation, 373
array, 7, 386
 allocating, 51
 array-wise expression, 100, 101,
 111
 assigning to length, 106
 bounds checking, 8, 95, 96, 98, 101,
 108
 during compilation, 108, 109
 comparison, 100, 110
 concatenation, 100, 111
 contiguous, 112
 conversion, 38, 109
 copy semantics, 9
 copying, 98, 101, 109
 creation, 8, 93
 dup, 110
 duplicating, 94
 dynamic, 93
 empty, 95
 expansion, 103, 104
 filling, 101
 fixed-size, 38, 107
 global, 107
 high-level, 126
 indexing, 50, 108
 initialization, 93, 107
 with literal, 94

iteration, 108
jagged, 112
length, 8, 95, 108
 changing during iteration, 75
length, 95, 106
literal, 39, 107
 and immutability, 39
 element type, 40
 length, 39
low-level, 126
multidimensional, 111, 136
 columns, 113
null, 95
of arrays, 52, 111
overlapping, 101
pass by value, 110
passing convention, 132
ptr property, 125
quick reference, 126
reallocation, 103, 104, 106
 in place, 104
representation, 98
resizing, 102, 103, 106
safety, 126
shape, 112, 113
sharing, 98
shrinking, 102, 103, 106
slice, 101, 102
slicing, 10, 50, 97, 98, 109
sparse, 378
static allocation, 107
statically sized, 9
stomping, 105
uninitialized, 107
 updating during iteration, 76, 77
array literal
 element type, 39
Artificial Intelligence, 131
ASCII, 118, 119, 124, 338
Asian writing systems, 119
asm, 89, 392
assembly language, 89
assert(false), 326
assert, 46, 47, 97, 316, 317, 325, 326
ASSERT_ALWAYS, 325
AssertError, 46, 316, 317
assertion, 287, 314, 316
assignment
 left-hand and right-hand side of,
 42
operator, 60
 user-defined, *see opAssign*
precedence, 60
associative array, 7, 8, 12, 15, 114, 115
 in, 56
 null, 114
 byKey, 117
 byValue, 117
 comparison, 116
 copying, 115
 dup, 115
 getting keys, 13
 indexing, 50
 insertion, 8
 iteration, 77, 116, 383
 keys, 117
 length, 114
 literal, 40, 114
 lookup, 8
 membership test, 56, 115
 order of elements, 77
 quick reference, 126
 reading and writing, 115
 remove, 116
 removing element, 116
 type, 114
 user-defined key type, 117
 values, 117
associativity, 58
 of assignment, 60
asynchronous message, 398
ATL, 152
atomic, 424, 426
access, 416

increment, 426
 operation, 412
 primitive, 426
 reads and writes, 413
 attribute, 96, 156, 165
auto, 11, 93, 107, 153
automatic
 conversions, 29
 expansion, 164
 memory management, 126
automaton, 33, 34
average, 20, 24
Average, 25
average, 159

B

back to the basics, 27
backslash, 34
 bank account example, 415
base class, 191
 Basic Multilingual Plane, 120
basic types, 29, 30
 Baxter, Bill, xxvii
 Bealer, Kevin, xxvii
binary code, 132
binary literal, 32
binary operator type, 45
binary resources, 37
binary search, 10
binary searching, 131
binary tree, 131
binding, 177, 242
binding contract, 314
bitfields, 47
bitwise
 AND, 58
 OR, 58
 XOR, 58
bitwise complement, 53
blit, 247
block statement, 82
block structure, 6

block transfer, 247
body, 317, 320
boilerplate code, 369
BOM, *see byte order mark*
bool, 29, 32
 Boucher, Travis, xxvii
bounds checking, 50, 51, 95, 96, 98, 101
braces, 66, 68, 82
break, 78
 labeled, 80
breakage, 95
 Brooks, Fred, 199
brute-force search, 142
build
 non-release versus release, 96
 release, 97
built-in type versus user-defined type, 365
byLine, 16
byte order mark, 337, 338
byte, 4, 29

C

C, 2, 4, 11, 14, 26, 29, 37, 42, 43, 50, 52, 58, 68, 103, 137, 139, 152, 188, 270, 272, 287, 299, 315, 317, 349, 351, 361, 395, 398
callbacks, 152
interfacing with, 359
tradition, 2
c (string literal suffix), 39, 123
C++, 2–4, 11, 15, 26, 29, 43, 50, 74, 86, 174, 180, 184, 198, 214, 226, 249, 272, 287, 299, 343, 349, 351, 395
interfacing with, 359
slicing, 26
C99, 30, 34
C#, 3, 11, 29, 198, 214, 226, 395, 417
cache, 393, 396
 dedicated, 396
cache effects, 84

- cache hierarchy, 414
cache miss bonanza, 112
call stack, 154, 382
calling convention, 359
calling method dynamically, 386
Calvin, 147
Cambrian explosion, 118
capacity, 392
Cardelli, Luca, 354
cas loop, 428
cas-based programming, 427
cas, 426
cascading if-else, 68
case sensitive, 210
case, 71, 79
Casinghino, Mike, xxvii
cast, 23, 45, 53, 193, 328, 369, 423
Castilla, Álvaro Castro, xxvii
casting, 23
catch site, 301
catch, 65, 81, 82, 302, 303, 305, 309, 317
causality, 396, 408, 413
chains of comparisons, 58
Chang, Richard, xxvii
char [], 16
 versus string, 17
char, 29, 39, 120, 121
character literal, 34
character type for string, 37
chef d'oeuvre, 13
child class, 191
chmod, 2
class, 15, 21, 51, 175, 177, 181, 183,
 188–190, 200, 212, 214, 219,
 221–223, 225, 226, 229, 230,
 234, 261, 269
anonymous, 226
layout, 359
operator overloading for, 383
parameterized, 233
reference semantics, 177
classic multithreading, 395, 396, 416,
 418
clear, 187, 188, 254
clearlsb, 430
clock rate, 393
closure, 153, 154
closures, 7
cluct, 246
Clugston, Don, xxvii
coalescing, 15, 104
code bloat, 139
code for “doing” versus “being”, 131
code point, 122
code shape, 138
code unit, 122, 123
coding standards, 202
coercion, 43
Cohen, Tal, 207
collateral exception, 307
collection, 380
combinatory, 140
come hell or high water, 85
comic strip, 147
comma expression, 60
command line, 1
 parameters, 22
command line arguments, 102
command prompt, 2
comments, 3
common type of two values, 60
compact code, 139
compare-and-swap, 426
 double-word, 426
comparison
 for equality, 56, 57, *see opEquals*
 for non-equality, 57
 for ordering, 58, *see opCmp*
 non-associativity of, 58
compilation model, 139
compile-time
 expression, 69
 function evaluation, 69

string manipulation, 47
compile-time constant, 108, 137, 188
compile-time enforcement, 405
compile-time evaluation, 171, 173
 limitations, 174
compile-time function evaluation, 349,
 388
compile-time introspection, 48
compile-time parameters, 383
compile-time selection, 69
compiler error messages, 141
compiler flags, 11
compiling programs, 2
complex number, 366
complexity, 139, 166
 of connectivity, 392
compound statement, 66
compress, 157
computational density, 392
computational power, 393
computing industry, 393, 394
computing power density, 392
computing resources, 391
concatenation, 55, 100
concurrency, 391, 394
 magic potion for, 426
 model, 394
concurrent applications, 431
concurrent execution, 395
concurrent imperative language, 391
concurrent programming, 290
Concurrentgate, 392
condition, 316
conditional compilation, 69
conditional operator, 39, 59
configuration file, 341
conflict in function calls, 147
conjunction, 333, 334
connectivity hierarchy, 392
conservative, 179
consistency, 206
`const`, 136, 287, 297, 299
constant parameter, 136
constants
 introducing, 4
constraint, 141
constructor, 24, 51, 181
 forwarding, 183
 overloading, 183
contention, 397
contiguity, 154
contiguous object, 230
contiguous portion of array, 97
contiguous storage, 132
`continue`, 78
 labeled, 80
contract, 329
contract in interface, 334
contract programming, 313–336
contractual requirement, 316
control, 202
control character, 34
control flow, 83, 302
conversion
 inefficiency, 374
 shortest path rule, 46
convertible, 48
copy elision, 249, 251
copy-on-write, 122
copying files, 406
core dump, 47
`core.gc`, 52
`core.memory`, 187
`core`, 361
correctness, 424
counting words, *see* example
coupling, 180
`cout`, 4
covariant return type, 196
CPU, 393–397, 416
critical section, 392, 395, 414
cross-module overloading, 146
crosstalk, 392
curly braces, 5, 66

custom float, 366

cyclops, 413

Cygwin, 2

D

d (string literal suffix), 39, 123

dangling reference, 180

Darwin awards, 155

data, 131

 common format, 139

 initializing, 137

 integrity, 413

 topology, 154

 visibility, 137

data block, 396

data bus, 393

data contention, 397

data hiding, *see* information hiding

data race, 392, 408

data sharing, 392, 395–397, 418, 431

 underworld of, 431

data structure, 14

database programs, 131

datum, 393

DBCS, *see* double-byte character set

dchar, 29, 39, 120, 121, 124

dead assignment elimination, 185

deadlock, 418, 424, 425

 cycle, 425

deallocated object, 187

debug, 361

debugger, 47

decimal literal, 32

declaration, 2, 70

decoupling, 199

decrement, 50, 378

default constructor, 182

default initialization, 178

default initializer, 30

default sharing, 424

default value, 184

default-share language, 398

default, 71, 79

deferred typechecking, 139

delegate, 41, 150, 382, 387

delete operator, 187

delete, 188

density, 393

density of integration, 392

dependency, 199

deprecated, 359

dereference, 52

dereference operator, 124

derived, 191

descendant, 191

descendant class, 191

design consistency, 214

desktop computer, 393

destructor, 86, 186–188, 195, 417

detective investigation, 393

deterministic finite automaton, 33

DFA, 33

diamond hierarchy, 229

dictionary, 8

Dilly, Stephan, xxvii

disambiguation, 143

discriminated union, 272

disjunction, 331

dispatch overhead, 198

distributed computing, 82

division, 54

 by zero, 54

division operation, 46

divisor, 171, 172

dmd, 340, 351, 352

do while, 73, 78

documentation, 316

documentation comments, 358

Domain-Specific Embedded Language,
47, 388

double-byte character set, 345

double, 29

doubly initialized object, 184

download hamlet.txt, 13

dramatis personae, 14
DSEL, 47, *see* domain-specific embedded language
`dstring`, 39, 121
dual-purpose class, 419
`dup`, 94, 110
duplicating array, 94
duplication, 180
dynamic array, 93
dynamic dispatch, 386
dynamic inheritance, 387
dynamic language, 384
dynamic memory allocation, 51
dynamic method, 385
dynamic polymorphism, 15, 26
dynamic scripting, 47
dynamic type, 192

E

eager copy, 122
ease of maintenance, 131
edit-run cycle, 2
effect, 135
effect for expressions, 66
effect visibility, 395
Effective Java, 207, 396
efficiency, 47, 56, 96, 104, 112, 198, 230, 392
Eiffel, 226, 314
electromagnetic field, 392
ellipsis, 159, 386
`else`, 67, 70
embedded language, 395
embedding binary resources, 37
emoticon, 413
empty array, 95
empty statement, 67
`empty`, 155, 157, 380
encapsulation, 199–203, 212
encryption, 82
endianness, 361
`ENFORCE`, 325

`enforce`, 217, 325, 326, 377
English, 119
entity, 177
`enum`, 240, 272–276
 base type, 274, 275
 `init`, 275
 `max`, 275
 `min`, 275
enumerated type, 72
eponymous, 281
equality, 56
equally specialized functions, 144
equational reasoning, 181
Erlang, 395, 397, 398
error
 at call site versus in implementation, 141
 handling, 88, 301, 313, 406
 isolation, 131
 message, 46, 140
`Error`, 302, 317, 319
escape rules for synchronized, 419
Euclid’s algorithm, 170
 ϵ , 118
evaluation order, 50
event variable, 392
event-driven logic, 398
example
 bank account, 415
 `bump`, 135
 counting words, 12
 `find`, 132
 frequency counting, 12
 heights in feet and centimeters, 3
 `makeHammingWeightsTable`, 83
 `stats`, 20
 `transmogrify`, 146
 triangular matrix, 111
 `underscoresToCamelCase`, 385
 vocabulary building program, 7
exception, 72, 82, 301, 357
 `uncaught`, 312

Exception, 302, 319, 326
exceptional path, 302
execution order, 357
exploit, 95
exponent, 34
exponential elbow, 394
exponential expansion, 393
exponential time, 166
export, 201–203, 261
expression
 cast, 53
 additive, 55
 address-of, 52
 bitwise
 AND, 58
 OR, 58
 XOR, 58
 bitwise complement, 53
 comma, 60
 comparison with screwdriver, 47
 compile-time, 69
 concatenation, 55, 100
 division, 54
 index, 11, 93
 indexing, 50
 logical
 AND, 59
 OR, 59
 multiplicative, 54
 negation, 53
 new, 51
 parenthesized, 46, 49
 power, 54
 primary, 49
 quick reference, 61
 remainder, 54
 shift, 55
 slice, 11
 statement, 5
 subtraction, 55
 unary minus, 53
 unary plus, 53

 with no effect, 66
expression statement, 65
expressive power, 139
expressiveness, 230
extern, 359

F

factorization, 173
factory, 22
Factory design pattern, 211
factory, 210
false, 32
fat pointer, 132, 134
fetch-and-add, 426
Fibonacci, 166
field initialization, 184
FIFO, 398, 426
Filali, Karim, xxvii
file, 337
file attributes, 202
file copying, 406
file inclusion, 2
file suffix, 337, 348
final class, 199
final switch, 65, 72, 73, 78
final, 25, 197–199, 220, 240
finally, 65, 81, 82, 84, 306, 311
find, 17
fire-and-forget, 395
first match rule, 303, 405
first pass through a function, 137
float, 29
floating-point, 58
floating-point arithmetic, 366
floating-point literal, 33, 34
foldl, 157
for, 65, 74, 78
forcing identical mutexes for owned objects, 422
foreach, 3, 9, 65, 74, 78, 94, 113, 116, 161, 380, 381
 on arrays, 75

with strings, 123
forehead slapping, 395
form follows function, 413
form follows structure, 6
formatted printing, 5
Fortin, Michel, xxvii
forwarding constructors, 183
fractal principles, 337
frame pointer, 150
french roast, 116
french vanilla, 116
frequency counting, *see* example
fresh symbol, 74
Friedl, J., 331
`front`, 155, 157, 380
function, 131, 149
 address of, 152
 array of, 152
 frame, 150
 higher-order, *see* higher-order function
 literal, 149
 nested, *see* nested function
 overloading, *see* overloading
 pointer to, 152
 signature, 315
 trailing parens in call, 156
 variadic, *see* variadic function
function call operator, 50
function literal, 14, 40, 50
 environment, 41
 syntax, 41
 type deduction in, 41
function, 41, 150
functional blocks, 392
functional language, 395, 397
functional programming, 290, 395
functional purity, 165
functional style, 288
functions
 arguments, 6, 13
 defining, 6

parameters, 6
fundamental types, 29

G

garbage collection, 52, 77, 154, 178, 180, 186–188, 195, 269
garbage collection, dangling reference, 269
`GC.free`, 187, 188
generality, 140
generated symbol, 74
generic argument, 149
generic function, 11
generic function arguments, 13
generic parameter, 149
`getchar`, 406
global guarantee, 391
global namespace, 341
global order of reads and writes, 414
global ordering of objects, 425
global scope, 341
global symbol access, 66
global variable, 398
global versus local connectivity, 392
glorified macro, 139
golden ratio, 131
`goto case`, 79
`goto default`, 79
`goto`, 15, 78, 82
 good use of, 79
graph, 180
greater than, 58
greatest common divisor, 170
grouping with `static if`, 69
guarantee through convention, 412
guesstimate, 203

H

hackish flavor, 429
halt instruction, 326
`Hamlet`, 12, 14
 statistics, 19

Hamming weight, 82
handshake, 396, 416
hard coding, 138
hard drive, 406
hardware crowd, 393
hardware developments, 392
hardware exception, 54
hardware industry, 391
hardware memory protection, 394
harmonic mean, 366
hashtable, 8, *see* associative array
haslrb, 430
haystack, 132, 141
head and tail of data, 420
header file, *see* module, summary
heap-allocated memory, 154
Held, David B., xxvii
Hello, world, 1
 in Italian, 115
 in Latin, 115
Helvensteijn, Michiel, xxvii
Helyer, Bernard, xxvii
Herlihy, Maurice, 426
hexadecimal literal, 32, 34
hexadecimal string literal, 36
hidden exit point, 302
hiding, 66, 67
high-level invariant, 398
higher-order function, 40, 148, 152, 153,
 314
highly parallel architecture, 392
hijacking, 147, 215
HLT, 47, 326
Hobbes, 147
homogeneous versus heterogeneous
 translation, 139–140, 235–236
honor system, 412, 421
House, Jason, xxvii
how to get free coffee, 415
Hu, Sam, xxvii
Hume, Thomas, xxvii

hurting the few and the virtuous
 as opposed to hurting the plebes,
 416

I

IDE, 1
IEEE 754, 4, 34
`if`, 6, 67
illusion of data sharing, 393
immutable sharing, 431
`immutable`, 4, 17, 37, 38, 83, 121, 123,
 287, 288, 290, 400, 401, 407,
 412
constructor, 293
conversions, 295
method, 292
 with array types, 38
imperative language, 397
implicit conversion, 46, 48, 140
implicit numeric conversions, 42
 rules, 43
import paths, 37
`import`, 2, 37, 84, 122, 146, 338–349,
 351, 352
 `public`, 344
 `static`, 347
 renaming, 347
 repeated, 2
 selective, 346, 347
impossibility results, 426
impure (for object-oriented languages),
 181
in place modification, 378
`in`, 7, 8, 56, 115, 135, 317, 320, 331
`include`, 37
incompatible function arguments, 141
increment, 50, 378, 412
indentation style, 5, 6
index expression, 93
indexed access with lists, 155
indexing, 50
indirection, 112

inefficiency, 139
infinite lifetime, 180
infinite loop, 407
infinite-precision number, 366
information, 199
information hiding, 199, 201, 337
inheritance, 24, 190, 191, 318, 329, 332
`init`, 7, 30, 51, 93, 106, 107, 120, 178,
 181, 244, 251, 270
initialization, 30, 186
 with `void`, 245
initialization order, 189
`inject`, 157
inorder traversal, 382
`inout`, 299
insertion and deletion in lock-free list,
 429
`int`, 4, 29
integral literal, 32
integration density, 393
integrity, 125
Intel x86, 89
interconnectivity, 392, 393
interesting times, 392
interface, 199, 212–214
 versus implementation, 212
`interface`, 21, 212, 214, 218, 227, 229
interleaving, 399
interlocked instructions, 426
interlocking, 395, 397
internal iteration, 381
internal logic, 324
interpreted language, 47
interpreter, 47
interpreting string, 47
interprocedural analysis, 421, 422
interval, 3
invariant, 314
`invariant`, 321, 334
inversion of control, 381
irreducible, 171
irreflexivity, 209, 210

`is`, 46, 100
 expression, 48
 for equality, 57
iteration, 3, 9, 100, 380
 internal, 381
 primitives, 380
 state, 382
iteration index, 76
iterator
 hierarchy, 157
Iterator pattern, 157

J

jack-of-all-trades, 386
Jacques, Robert, xxvii
jagged array, *see* array, jagged
Java, 3, 11, 29, 198, 214, 226, 272, 395,
 417
java, 116
JavaScript, 386
junk mail, 405

K

K Desktop Environment, 329
K&R, 82, 406
K&R C, 315
Kamm, Christian, xxvii
Keep, Daniel, xxvii
Kegel, Mark, xxvii
Kelly, Sean, xxvii
Kernighan, Brian, 82, 315
keywords, 31
Khesin, Max, xxvii
kind, 49
Kjaeraas, Simen, xxvii
Klingon, 133
Koeninger, Cody, xxvii
Koroskin, Denis, xxvii
Kyllingstad, Lars, xxvii

L

- label, 78, 221
labeled break, 80
labeled continue, 80
lambda function, 14, 40, 369
lambda functions, 7
lamp genie, 53
language abstraction, 391
large-scale development, 288
large-scale modularity, 337
Latin-derived writing systems, 119
laundering types, 299
lazy evaluation, 326
least significant bit, 429
least significant byte, 43
left-to-right evaluation, 50
legally acquired movie, 410
length, 8, 51, 95
Lesik, Igor, xxvii
less specialized function, 144
less than, 58
let me explain, 126
Letuchy, Eugene, xxvii
lexical order, 189, 357
lexical scope, 66
lifetime, 252
 - finite versus infinite, 239

LIFO, 86
limit applicability of function, 141
limiting function signature, 142
linear algebra, 239
linear congruential generator, 169, 172
 - periodicity, 169

linear search, 131, 148, 154
linguistic ability, 143
linked list, 132
linker, 133
Linux, 89
Liskov Substitution Principle, 329
Lisp, 7, 14
list, 180

list format, 139
literal, 46, 365
 - array, 39
 - binary, 32
 - character, 34
 - floating-point, 33, 34
 - hexadecimal, 32, 34
 - hexadecimal string, 36
 - integral, 32
 - octal, 32
 - string, 35, 38, 84
 - string, 122
 - suffix, 32
 - suffix of literal strings, 39, 123
 - type, 32, 33, 37

literals, 4
local alias, 147
local environment, 154
local versus global connectivity, 392
lock, 425
 - tyranny of, 426

lock-based
 - discipline, 424
 - program, 415
 - programming, 392, 418, 431
 - synchronization, 414, 418, 422

lock-free, 427
 - stack, 427

locking-intensive, 423
logic bug, 325
logical address, 394
logical AND, 59
logical character, 122
logical OR, 47, 59
logically deleted pointer, 429
long-range coupling, 180
long, 4, 29
lowering, 74, 85, 87, 379
 - of operators, 366

lvalue, 42, 52, 80, 101, 102, 135, 257, 265
 - defined, 42
 - versus rvalue, 42

M

Månsson, Pelle, xxvii
macro, 139
magic, 239, 379
mailbox, 404, 405, 410
 crowding, 404, 410
main memory, 396
main, 1, 2, 102, 357
 command line parameters, 22
 return type of, 1
mainframe, 394
mainstream hardware, 395
maintainability, 193, 194
maintenance, 73
`malloc`, 52, 188
mangling, 359
mantissa, 34
manual memory management, 52
Masahiro, Miura, xxvii
masking, 66, 67, 81
massive concurrency, 418
math-fu, 144
`math.h`, 315
mathematical purity, 395
matrix, 239, 366
Matthews, Tim, xxvii
maximum, 20
maximum mailbox size, 410
Maya veils, 416
member access, 49
member initialization, 25
`memcpy`, 245
memory
 architecture, 393
 density, 393
 hierarchy, 393
 protection, 394
 speed, 393
 virtualization, 394
memory allocator, 104
 overhead, 112

memory architecture, 395
memory barrier, 414
memory bus, 406
memory isolation, 398
memory recycling, 188
memory region, 51
memory safety, 126, 180, 188, 355
memory subsystem, 396
memory-unsafe feature, 125
message passing, 395–398, 431
 versus memory sharing, 391
Message Passing Interface, 395, 398
message pump, 398
`MessageMismatch`, 402
method, 182, 190
 not defined, 385
method invocation syntax, 156
metric, 202
Meyer, Bertrand, 175, 211, 314
Meyers, Scott, xxvii, 202
MFC, 152
Microsoft Windows API, 329
migration, 359
Milewski, Bartosz, xxvii, 246
miniaturization, 392
minimum, 20, 158
misaligned memory access, 267
`mixin`, 46, 47, 65, 82, 83, 282, 284, 386,
 388
 in operator definition, 368
`mixin expression`, 276
ML, 154
modern machine, 431
modern multithreading, 416, 418
Modula-3, 355
modular development, 337
modularity, 14, 23, 67, 84, 131, 199
module, 2
 safe, 96
 safety, 97
 system, 96
 trusted, 96

- module, 133, 337, 348, 349, 351
 - constructor, 356, 357
 - destructor, 357
 - initialization order, 358
 - looking up names in, 341
 - searching roots, 340, 351
- modules
 - initialization order, 189
- modulus, 54, 171
 - floating-point, 54
- Mohamed, Fawzi, xxvii
- monomorphic, 15
- Moore's law, 392
- more specialized function, 144, 145
- most significant byte, 43
- move, 251
- moving context, 146
- moving sofa constant, 39
- MPI, *see* Message Passing Interface
- multidimensional array, *see* array, multidimensional
 - tidimensional
- multiple inheritance, 226, 230
- multiple subtyping, 230
 - overriding methods with, 231
- multiplication, 54
- multiplicative expression, 54
- multiport memory access, 396
- multiprocessor, 396, 414
- multithreaded application, 396
- multithreading, 395
- mutable data, 397
- mutation, 165, 167, 395
- mutex, 392, 395, 414, 415, 423, 424
 - acquiring in the same order, 425
 - ham-fisted, 426
 - modular acquisition, 425
- mutual exclusion, 431
- mutually referential structures, 26
- Mythical Man-Month, The, 199
- N**
- name hiding, 66, 67
- name lookup, 176, 341
- name resolution, 66
- namespace, 23
- naming convention, 385
- NaN, *see* Not a Number
- natural number, 53
- needle, 132, 141
- negation, 53
- nested structs, 261
- nested class, 51, 222
 - in function, 223
 - static, 225
- nested function, 150
- nested try/finally, 86
- network drive, 410
- network service, 406
- new, 49, 51, 93, 94
 - considered harmful, 211
 - expression, 176
 - placement, 52
- Newcomer, Ellery, xxvii
- newline in string literal, 36
- Niebler, Eric, xxvii
- nomenclature, 191
- non-associativity, 58
- non-blocking linked-list, 429
- non-equality, 57
- non-local effect, 17
- non-release build
 - versus release build, 96
- Non-Virtual Interface, 213–215, 335
- nonsensical argument, 141
- nonzero, 378
- nonzero bits in a byte, 82
- nonzero value, 46
- Not a Number, 54, 58, 181, 354
- nothrow, 168, 315
- null, 56, 114, 124, 178, 192, 206
- numeric code, 366
- numeric relation, 144
- numeric types, 4
 - signed versus unsigned, 4

NVI, *see* Non-Virtual Interface

O

object, 177

branding, 185

construction, 184

life cycle, 181

location in memory, 178

tear-down, 188

 tear-down sequence, 187

object orientation, 20

object serialization, 211

object-oriented programming, 175, 181

`object.di`, 276

`Object`, 203, 341

`object`, 341

octal literal, 32

octonion, 366

offer more, 329

`OnCrowding`, 411

OOP, *see* object-oriented programming

`opAdd`, 368

`opApply`, 381, 382

`opAssign`, 256, 257, 376

`opBinary`, 366, 371, 383

`opBinaryRight`, 371, 373

`opCast`, 369

`opCmp`, 117, 209, 210, 256, 375

`opDispatch`, 384–386

`opDollar`, 379

Open-Closed Principle, 20, 211

`opEquals`, 205–209, 256, 258, 259, 375

operator, 366

 assignment, 60

 precedence, 60

 binary

 type of, 45

 conditional, 39, 59

 function call, 50

 unary

 type of, 45

versus function call, 366

operator overloading, 366

 \$, 379

 address, 388

 ambiguity error, 371

 binary, 371

 defined in terms of assignment,
 377

 comma, 388

 commutativity, 373

 comparison, 375

 conversion, 370

`foreach`, 380, 381

 identity test, 388

`if`, 370

 in classes, 383

 in-place, 376, 377

 index, 377

 logical conjunction, 388

 logical disjunction, 388

 lowering, 366

 overloading, 373

 postdecrement, 369

 postincrement, 369

 predecrement, 369

 preincrement, 369

 quick reference, 388

 slicing, 379

 ternary operator, 370, 388

`typeid`, 388

 unary, 367

operators, 4, 42

`opHash`, 117

`Ophelia`, 14, 19

`opIndex`, 377, 378

`opIndexAssign`, 378

`opIndexUnary`, 378

`opMul`, 368

`opOpAssign`, 376

`opSlice`, 379

`opSliceAssign`, 379

`opSliceOpAssign`, 379

`opSliceUnary`, 379

- opSub, 368
- optical disk, 406
- optimization, 414
- opUnary, 367
- order of execution, 86
- order of precedence, 42
- order of top-level declarations, 146
- ordering comparison, 58
- organic operation, 428
- orphaned method, 193
- OSX, 2
- out of order execution, 395
- out-of-band message, 409
- out-of-bounds access, 95
- out, 7, 136, 319, 320, 332, 333
- outer, 222, 232
- overflow, 53
- overflow bit, 373
- overloading, 142, 143
 - cross-module, 146, 343
 - food chain, 143
 - rules, 145
 - too lax versus too tight, 143
- override, 25, 191, 193, 194, 198, 220, 232
- owned object, 423
- owned subobject problem, 422
- OwnedFailed, 409
- owner thread, 407
- owner-owned relationship, 422
- OwnerFailed, 410
- ownership graph, 409
- ownership of objects, 421
- OwnerTerminated, 408
- P**
- package, 202
- package, 200, 202, 203, 261, 337
- padding, 267
- palindrome, 102
- parallel communication, 392
- parallel execution, 165
- parallel kernel, 394
- parallelization, 395
- parameter list, 138
- parameter tuple, 161, 163
- parameter type tuple, 161
- parameterized class, 233
- parameters
 - compile time vs. run time, 11
- parent class, 191
- parenthesized expression, 46, 49
- Parker, Mike, xxvii
- Parnas, David L., 200, 314
- Parnell, Derek, xxvii
- partial order, 145, 209
- partial ordering, 144, 145
 - of functions, 144
- Pascal, 4, 38
- pass by reference, 400
- pass by value, 110, 132
- pass by value and reference, 6
- pattern matching, 131
 - with receive, 403
- peeling braces with static if, 69
- Pelletier, Jeremie, xxvii
- performance, 96
- Perl
 - version 6, 58
- Perlis, Alan, 297
- persona, 15
- PersonaData, 16–18
- Phobos, 361
- physical address, 394
- physical character, 122
- Pierce, Benjamin, 175
- pipeline, 396
- placement new, 52
- platform dependencies, 360
- pointer, 52, 56, 124, 412
 - arithmetic, 125
 - dereference, 52
 - indexing, 50
- Russian roulette, 125

- slicing, 51
unchecked, 125
- pointer to function or delegate, 50
- Polish democracy, 40
- Polonius, 15
- polymorphism, 180, 181, 240
- popFront, 155, 157, 380
- portable digital assistant, 393
- postblit constructor, *see* `this(this)`
- postcondition, 314, 319, 332
- postdecrement, 50, 378
- postincrement, 50, 378
- `pow`, 6
- power operator, 54
- power-to-weight ratio, 131
- precondition, 314, 317
- predecrement, 53, 378
- predicate, 148, 316
- preincrement, 53, 378
- premature optimization, 198
- primality, 173
- primary exception, 308
- primary expression, 46, 49
- prime factors, 171
- primitive expression, 31
- `printf`, 4, 47, 287
- priority message, 409
- priority queue, 398
- `PriorityMessageException`, 410
- `prioritySend`, 409
- private state, 166, 167
- `private`, 137, 200–203, 214–217, 261, 323, 329
- process, 394, 395, 397
 isolation, 394
- processor state, 395
- producer-consumer protocol, 406
- program correctness, 313
- program name, 103
- program state, 152
- program termination, 400
- programming discipline, 131
- programming model, 393
- programming paradigm, 175, 394
- programming practices, 11
- programming-by-honor-system, 412
- proper subtype, 191
- property access syntax, 156
- protected memory, 37
- `protected`, 200–203, 216, 217, 232, 323
- pseudo method, 156
- pseudo-member notation, 156
- pseudorandom, 169
- `ptrdiff_t`, 278
- `public`, 201–203, 212, 216, 217, 232, 261, 323
- pure functional languages, 26
- pure object-oriented languages, 26
- `pure`, 165, 167, 315
- `putchar`, 406
- Python, 2, 6, 58, 343
- Q**
- `Qt`, 152
- qualifier, *see* type qualifier
 `shared`, 397
- qualifying a function name, 147
- qualifying member name, 195
- quaternion, 366
- Queen, 19
- queue, 398, 404, 409
- quick reference
 expressions, 61
 statements, 89
- quote, 36
- quote character, 34
- quoted string literal, 35
- R**
- race condition, 415
- races, 431
- RAII, *see* Resource Acquisition Is Initialization
- RAM, 393

- random number generator, 169
range, 132, 157, 380
 input, 157
range violation, 115
`RangeError`, 50, 51
raw data definition, 36
`rdmd`, 2, 133
read-modify-write, 106, 412
read-modify-write operations, 415
read-only, 37
read, 328
`readf`, 23
`readText`, 328
`real`, 29, 413
rebinding, 178
`receive`. `Tuple`, 404
`receive`, 403–405, 408
`receiveOnly`, 402, 404
`receiveTimeout`, 404
recursive definition, 191
recursive function, 11
red-black tree, 131
`reduce`, 157, 158
reductio ad absurdum, 210
redundant inheritance paths, 227
`ref`, 6, 7, 9, 76, 94, 108, 110, 113, 135,
 136, 156, 243, 381
 and conversions, 76
 giving access, 378
refactoring, 67
reference implementation, 351, 353,
 356
reference semantics, 15, 26, 177, 180,
 239, 242, 246
reference versus object, 177
referential structure, 180
reflexive, 144
reflexivity, 206, 209
regex, 15, 17
regular expression, 15, 331
relational algebra, 131
relative primality, 170
relaxed memory model, 416, 418
release mode, 326
reliability engineering, 313
relocatable objects, 249
remainder, 54
removing method at runtime, 386
renaming in `import`, *see import, renaming*
reordering, 414, 416
reordering of field access, 421
require less, 329
resistance, 392
Resource Acquisition Is Initialization,
 310
resource disposal, 188
resource leak, 310
return, 81, 319
reusability, 139
reverse video, 124
reversed turtles, 190
rewriting
 for overloaded operators, 366
rigidity, 139
Ripolles, Pablo, xxvii
Ritchie, Dennis, 82, 315
Roberts, Brad, xxvii
root, 204
run-of-the-mill machine, 396
runaway process, 394
runnable thread, 396
runtime check, 313
`rvalue`, 42, 101, 102, 135, 182, 249, 251,
 257, 265, 266
`rvalue` versus `lvalue`, *see lvalue versus rvalue*
Rynn, Michael, xxvii

S

- safe interface, 96
safe module
 versus system module, 96

safety, 95, 96
 memory, *see* memory safety
 of casts, 53
SafeD, 126, 391
Sailor, 19
Savas, Foy, xxvii
scaffolding, 131
Scala, 230
scalability, 84, 86, 392, 431
Schardt, Christof, xxvii
scheduler, 394
Schveighoffer, Steve, xxvii
`scope(exit)`, 84, 85
`scope(failure)`, 87
`scope(success)`, 86
`scope`, 65, 84, 86, 240, 312, 366
scoped lifetime, 252
scoped locking idiom, 417
scoped object lifetime, 417
scoping with `static if`, 69
screwdriver compared with expression, 47
script, 2, 8
scripting, 387
scrubbing input, 327
search engine, 131
searching, 131
selective implementation of methods, 217
self-referential types, 26
semantically equivalent code, 74
semantics
 compiler awareness of, 365
semaphore, 395
seminal work, 426
send, 402–404
separate compilation, 422
separation of concerns, 337
sequence of statements, 66
sequential consistency, 414
serial communication, 392
serialization of access, 415
 setlsb, 430, 431
 setMaxMailboxSize, 410
 setMutex, 422
 setOwner, 407
Shakespeare, William, 13
shallow copy, 9, 246
shared address space, 395
shared class, 427
shared data, 413, 415, 416
shared memory, 396
shared resource, 180
shared struct, 427, 428
shared, 287, 397, 398, 400, 401, 411, 412, 414, 419, 420, 423, 431
sharing of data between threads, 391
shebang, 338
shebang notation, 2
shell, 1
shift expression, 55
shift happens, 55
short-circuit, 58, 331
short, 4, 29
Shropshire, Benjamin, xxvii
shutdown protocol, 407
signal propagation speed, 393
signals and slots, 152
signature, 138, 141, 315
signature constraint, 141, 143, 367
signed, 74
silicon oxide, 393
Simcha, David, xxvii
simplicity, 1
single statements, 5
single-processor, 397
singly linked list, 308
singly-linked list, 427
 lock-free, 429
sitcom cliché, 22
`size_t`, 276
`sizeof`, 276
skip list, 131
slack space, 104

slice, 381
slice of an array, 98
slicing, 11
slicing arrays, 50
smörgåsbord, 191
Smalltalk, 386
smart phone, 406
software crowd, 393
software engineering, 337
software industry, 391
something completely different, 384
sorting, 13
space-efficient storage, 378
sparse array, 378
spawn, 399, 402, 407
special symbols, 31
specialized hardware, 395
specialized versions of functions, 139
specification, 313, 314, 316
specification check, 328
speed gap between processor and
 memory, 393
speed of light, 393
split, 17
splitter, 8
SQL, 47
sqrt, 315
squint, 140
St. Jack, Graham, xxvii
Stachowiak, Tomasz, xxvii
stack, 233, 395
 lock-free, 427
stack overflow, 107
stack unwinding, 309
stack variable, 398
standard deviation, 158
standard error, 357
standard library, 361
startsWith, 15
statement
 asm, 89
 block, 82
 break, 78
 compound, 66
 continue, 78
 deltas from existing languages, 65
 do while, 73
 empty, 67, 73
 expression, 65
 for, 74
 foreach, 74
 on arrays, 75
 goto, 78
 if, 67
 mixin, 82
 quick reference, 89
 return, 81
 scope, 84
 static if, 68
 switch, 71
 synchronized, 88
 throw, 81
 top level, *see top-level statement*
 try, 81
 while, 73
 with, 80
static class constructor, 189
static class destructor, 189, 190
static else, 70
static if, 48, 65, 68, 278
static import, *see import, static*
static information, 196
static this(), 188
static type, 192
static versus dynamic, 387
static, 137, 151, 176, 182, 196, 252,
 260, 263
 obligatory joke about overuse of,
 345
statically scoped symbol lookup, 139
statistical encoder, 120
std.algorithm, 15, 157, 158, 178, 251
std.bitmanip, 47
std.c.fenv, 58

`std.concurrency`, 410, 412, 413
`std.contracts`, 325, 327
`std.conv`, 102, 161, 317
`std.file`, 328
`std.random`, 94, 174
`std.range`, 158
`std.regex`, 331
`std.stdio`, 2, 400
`std.typecons`, 163
`std.utf`, 122
`std.variant`, 386, 405
`std`, 361
`stdout`, 400
Stewart, Robert, xxvii
STL, 74, 157
stomping, 105
storage class, 6, 137
straw, 133
stream, 380
strict weak order, 209
`stride`, 122
string
 character type of literal, 39
 compiling into code, 83
 concatenation, 22
 fixed-size, 38
 length of literal, 38
 literal, 35
 literal suffix, 39, 123
 newline in literal, 36
 quoted literal, 35
 WYSIWYG literal, 35
`string`, 17, 37, 39, 118, 121, 122, 401
 copying, 17
 length, 38
`stringof`, 163
`struct`, 15, 17, 80, 240–247, 252–256,
 258–263, 266, 267, 269
construction sequence, 245
constructor, 244
field layout, 267
forwarding constructor, 245
initialization, 17
methods, 255
 vs. `class`, 26
subclass, 191
substitutability, 329
subtraction, 55
subtype, 191, 192
 proper, 191
sum, 158
`super`, 194–195
`super`, 31
superclass, 191
supertype, 191
surgery on code, 20, 211
surprising behavior
 of unary `-`, 53
surrogate pair, 124
Susie, 147
Sutter, Herb, 213, 398
`swap`, 178
`swap`, 178
`switch`, 65, 71, 78, 79
symbol, 30, 339–348
 generated by compiler, 74
 lookup
 at module scope, 30
 table, 2
 visibility, 69, 80
symbol lookup, 337
symmetry, 206
synchronization, 415
 for an infinite number of threads,
 426
 object, 414
 primitive, 395, 426
 wait-free, 426
`synchronized class`
 field typing, 419
 rules, 419
`synchronized`, 88, 418, 419, 424, 425,
 431

protection against races
 local aspect, 419, 420
 temporary aspect, 419
semantics, 419
tail sharing, 420
 with multiple arguments, 425
syntactic sugar, 366
syntactically valid type, 48
system-fu, 2
system-level access, 96
system-level language, 89
system-level programmer, 96

T

table lookup, 83
tag-based switching, 72
tagged union, 272
tail call elimination, 12, 167
task switching, 394
tear-down, 187
Teigen, Knut Erik, xxvii
telephony application, 395
template, 279, 281, 282
tension between efficiency and static
 verifiability, 186
tensor, 366
termination of multithreaded applica-
 tion, 407
terminology, 191
ternary operator, 74
test-and-set, 426
test-driven, 133
testing, 133
text processing, 8
text, 317
textual inclusion, 84
theorem of structure, 6
`this()`, 240
`this(this)`, 240, 245, 247, 249–251,
 253, 273
 rationale, 249
`this`, 31, 182, 196

`thisTid`, 402
Thompson, Ken, 119
thread, 137, 394, 395, 397
 starting, 399
thread blocking, 425
thread id, *see* tid
thread isolation, 391, 397, 400
thread priority, 415
thread queue, 410
thread scheduler, 396
thread startup, 188
thread termination, 407
thread-local storage, 107, 398
threading amenities, 431
threading API, 407
`throw`, 81, 301, 302, 309
`Throwable`, 302, 307, 308
thumb drive, 406
tid, 401
`Tid`, 401, 402
tightest common type, 59
time invariance, 395
time quantum, 394
time sharing, 394, 396
time slicing, 397
time-sliced threads, 414
timer interrupt, 394
TLS, *see* thread-local storage
`to`, 16, 17, 83, 102, 161
`toHash`, 205
`tolower`, 17
top braces, 74
top level declaration, 339
top-level statements, 1
`toString`, 205
trailing `else`, 68
transactional creation of a file, 88
transactional semantics, 168
transcendental function, 366
transcoding, 124
transitive, 144
 for shared, 412

transitivity, 136, 206, 209, 210, 290
 of equality with zero, 209
 of sign, 209
 transitivity rule, 427
 transitory state, 152, 166, 167
 translation, 139
 tree, 180, 382
 tree container, 261
 tree iteration, 382
 trie, 131
 true, 32
 truncation, 54
 trusted module, 96
 try, 65, 81, 82, 84, 303, 366
 tuple, 163
Tuple, 163, 402
tuple, 163
 two's complement, 53
 type
 conciliation, 74
 enumerated, 72
 for operators, 45
 of literal string, 37
 self-referential, *see* self-referential types
 type constructor, 26
 type deduction, 11, 14, 138, 140, 149
 type erasure, 235
 type inference, 4, 29, 39
 type name syntax, 365
 type parameter, 138, 140
 type parameter, 10
 type parameterization, 139
 type parameters, 234
 type qualifier, 38, 287
 composition rules, 299
 propagating to result, 299
 syntax, 291
 type system, 397, 424
 type system integrity, 54
 typeid, 32, 37
 typeof, 37, 141, 163, 276

U

ubyte, 4, 29, 407
 UCS-2, 120
 uint, 4, 29
 ulong, 4, 29
 unary
 minus, 53
 plus, 53
 unary operator, 367
 unary operator type, 45
 unbound, 179
 unbounded amount of code, 202
 unbridled confidence, 427
 unchecked cast, 54
 underscore, 30
 in numeric literals, 32
 undue aliasing, 180
 unfortunately, 392
 Unicode, 34, 118–122, 338
 code point, 118, 119
 code unit, 119
 Consortium, 118
 encoding, 118, 121
 misconception about, 118
 specific spelling, 118
 variable-length encoding, 119
uniform, 94
uniformity, 139
 uninitialized data, 107, 185
 union, 240, 270–272
 uniprocessor, 414
 unit of compilation, 337
 unittest, 11, 133, 138, 256, 361
 universal characters, 30
 universality results, 426, 427
 Unix, 2, 340
 unlocking, 417
 unordered functions, 144
 unprotected sharing, 398, 416
 unqualified access to shared data, 416
 unsafe constructs, 107, 110

unsigned, 74
unsigned shift, 55
unsigned type, 53
unspecified value, 55
unstated assumption, 397
untrusted format strings, 5
untyped address, 152
untyped memory, 184
unwashed masses, 167
unwieldy code, 86
user input, 325, 329
ushort, 4, 29
UTF, 118, 121, 122, 124, 328, 333
UTF-8, 30, 36, 37, 118–121, 123, 338
 backward iteration, 120
 properties, 119
 synchronization, 120
UTF-16, 30, 36, 118, 120, 123, 124, 338
 criticism against, 120
 high surrogate area, 120
 low surrogate area, 120
UTF-32, 30, 36, 118, 120, 123, 338
`UtfException`, 328

V

value declaration, 74
value object, 177
value range propagation, 29, 43, 45
value semantics, 26, 178, 181, 242, 246
value type, 240
value versus reference, 15, 25
varargs, 315
variable-length encoding, 120
variadic function, 159
 heterogeneous, 160
 homogeneous, 159
variadic functions, 5
`Variant`, 386, 405
vector, 366
vector operation parallelization, 394
verboten, 400
`VERIFY`, 325

version control, 202
version, 360
virtual function dispatch, 72
virtual table, 229
Vlăsceanu, Cristian, xxvii
vocabulary, 8, 143
vocabulary building program, *see example*
`void`, 29, 107
 as initializer, 185, 245
Voltaire, 343
Voltemand, 20

W

`w` (string literal suffix), 39, 123
Wadler, Philip, 313
wait-free synchronization, 426
`walkLength`, 333
`wchar`, 29, 39, 120, 121
Wegner, Peter, 83
werewolf, 5
`while`, 73, 78
whitespace, 6, 8
Windows, 2, 89, 340, 356
`with`, 80
`writeln`, 4
`writeln`, 2, 4, 68, 81, 159
 atomicity of, 399
`wstring`, 39, 121
WYSIWYG string literal, 35, 36

X

x86, 89

Y

Y2K, 202
`yacc`, 47
you've been warned, 97

Z

zero-based, 50
Zolman, Leor, xxvii