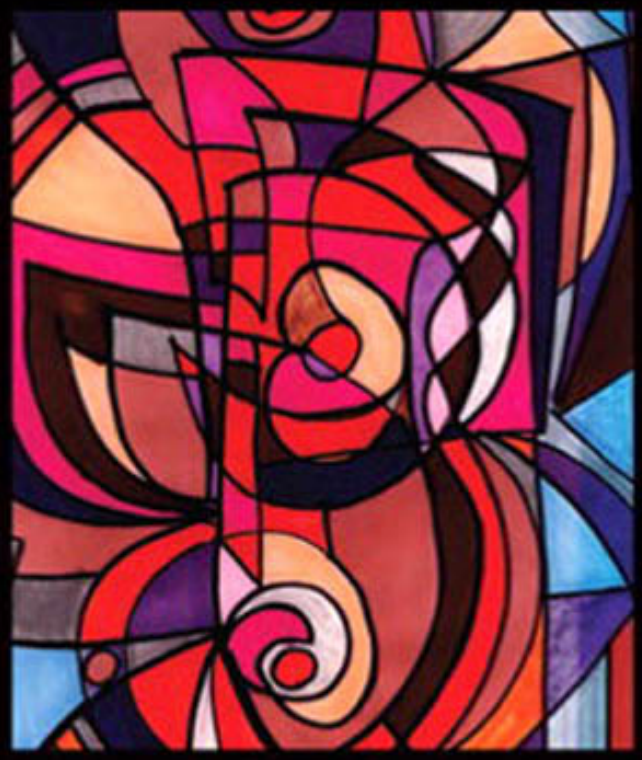
FOREWORDS BY WALTER BRIGHT AND SCOTT MEYERS



The D Programming Language

Andrei Alexandrescu

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

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

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 enforcible 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 justfy 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 multipurpose 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, Àlvaro 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 Månsson, 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, Cristian Vlăsceanu, and Leor Zolman.

Andrei Alexandrescu
Sunday, May 2, 2010

This page intentionally left blank

Chapter

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 tradeoffs 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 concurrency[1] 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 edition[3] [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 multithreaded 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 ‹a_1›, ‹a_2›, ..., ‹a_n›. The number of arguments n and their types must match `fun`'s signature, that is, the call `fun(`‹a_1›, ‹a_2›, ..., ‹a_n›`)` must be correct. This check is done at compile time. `spawn` creates a new execution thread, which will issue the call `fun(`‹a_1›, ‹a_2›, ..., ‹a_n›`)` 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 ‹Msg_1› and ‹Msg_2› with ‹Msg_2›'s handler coming after ‹Msg_1›'s in the `receive` call, `receive` makes sure that ‹Msg_2› is *not* convertible to ‹Msg_1›. If it is, that means ‹Msg_1› will match messages of type ‹Msg_2› so compilation of the call is refused. In the example above, the check fails when ‹Msg_1› is `long` and ‹Msg_2› 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 provides 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 `cast`s 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 `int`s 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) {
      ... // Same as for Stack.push
   }

   shared(T)* popFront() {
      ... // Same as for Stack.pop
   }
}
```

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

B

C

D

E

F

I

J

K

L

M

N

P

Q

R

S

T

U

V

W

X

Y

Z