contents

preface *xiii*
acknowledgments *xv*
about this book *xvii*
about the author *xx*
about the cover illustration *xxi*

1 ***Hello, world of concurrency in C++!*** ***1***

1.1 What is concurrency? 2

Concurrency in computer systems 2 ▪ Approaches to concurrency 4 ▪ Concurrency vs. parallelism 6

1.2 Why use concurrency? 7

Using concurrency for separation of concerns 7 ▪ Using concurrency for performance: task and data parallelism 8 When not to use concurrency 9

1.3 Concurrency and multithreading in C++ 10

History of multithreading in C++ 10 ▪ Concurrency support in the C++11 standard 11 ▪ More support for concurrency and parallelism in C++14 and C++17 12 ▪ Efficiency in the C++ Thread Library 12 ▪ Platform-specific facilities 13

1.4 Getting started 13

Hello, Concurrent World 14

2 *Managing threads 16*

2.1 Basic thread management 17

Launching a thread 17 ▪ Waiting for a thread to complete 20 Waiting in exceptional circumstances 20 ▪ Running threads in the background 22

2.2 Passing arguments to a thread function 24

2.3 Transferring ownership of a thread 27

2.4 Choosing the number of threads at runtime 31

2.5 Identifying threads 34

3 *Sharing data between threads 36*

3.1 Problems with sharing data between threads 37

Race conditions 38 ▪ Avoiding problematic race conditions 39

3.2 Protecting shared data with mutexes 40

Using mutexes in C++ 41 ▪ Structuring code for protecting shared data 42 ▪ Spotting race conditions inherent in interfaces 44 Deadlock: the problem and a solution 51 ▪ Further guidelines for avoiding deadlock 53 ▪ Flexible locking with std::unique_lock 59 ▪ Transferring mutex ownership between scopes 61 ▪ Locking at an appropriate granularity 62

3.3 Alternative facilities for protecting shared data 64

Protecting shared data during initialization 65 ▪ Protecting rarely updated data structures 68 ▪ Recursive locking 70

4 *Synchronizing concurrent operations 72*

4.1 Waiting for an event or other condition 73

Waiting for a condition with condition variables 74 Building a thread-safe queue with condition variables 76

4.2 Waiting for one-off events with futures 81

Returning values from background tasks 82 ▪ Associating a task with a future 84 ▪ Making (std::)promises 87 ▪ Saving an exception for the future 89 ▪ Waiting from multiple threads 90

4.3 Waiting with a time limit 93

Clocks 93 ▪ Durations 94 ▪ Time points 96 ▪ Functions that accept timeouts 98

4.4 Using synchronization of operations to simplify code 99

Functional programming with futures 99 ▪ Synchronizing operations with message passing 104 ▪ Continuation-style concurrency with the Concurrency TS 108 ▪ Chaining continuations 110 ▪ Waiting for more than one future 114 Waiting for the first future in a set with when_any 115 Latches and barriers in the Concurrency TS 118 ▪ A basic latch type: std::experimental::latch 118 ▪ std::experimental::barrier: a basic barrier 120 ▪ std::experimental::flex_barrier—std::experimental::barrier's flexible friend 121

5 *The C++ memory model and operations on atomic types 124*

5.1 Memory model basics 125

Objects and memory locations 125 ▪ Objects, memory locations, and concurrency 126 ▪ Modification orders 127

5.2 Atomic operations and types in C++ 128

The standard atomic types 128 ▪ Operations on std::atomic_flag 132 ▪ Operations on std::atomic<bool> 134 Operations on std::atomic<T>: pointer arithmetic 137 Operations on standard atomic integral types 138 The std::atomic<> primary class template 138 Free functions for atomic operations 140*

5.3 Synchronizing operations and enforcing ordering 142

The synchronizes-with relationship 143 ▪ The happens-before relationship 145 ▪ Memory ordering for atomic operations 146 Release sequences and synchronizes-with 164 ▪ Fences 166 Ordering non-atomic operations with atomics 168 ▪ Ordering non-atomic operations 169

6 *Designing lock-based concurrent data structures 173*

6.1 What does it mean to design for concurrency? 174

Guidelines for designing data structures for concurrency 175

6.2 Lock-based concurrent data structures 176

A thread-safe stack using locks 176 ▪ A thread-safe queue using locks and condition variables 179 ▪ A thread-safe queue using fine-grained locks and condition variables 183

6.3 Designing more complex lock-based data structures 194

Writing a thread-safe lookup table using locks 194 ▪ Writing a thread-safe list using locks 199

7 *Designing lock-free concurrent data structures 205*

7.1 Definitions and consequences 206

Types of nonblocking data structures 206 ▪ Lock-free data structures 207 ▪ Wait-free data structures 208 ▪ The pros and cons of lock-free data structures 208

7.2 Examples of lock-free data structures 209

Writing a thread-safe stack without locks 210 ▪ Stopping those pesky leaks: managing memory in lock-free data structures 214
Detecting nodes that can't be reclaimed using hazard pointers 218
Detecting nodes in use with reference counting 226 ▪ Applying the memory model to the lock-free stack 232 ▪ Writing a thread-safe queue without locks 236

7.3 Guidelines for writing lock-free data structures 248

Guideline: use std::memory_order_seq_cst for prototyping 248
Guideline: use a lock-free memory reclamation scheme 248
Guideline: watch out for the ABA problem 249 ▪ Guideline: identify busy-wait loops and help the other thread 249

8 *Designing concurrent code 251*

8.1 Techniques for dividing work between threads 252

Dividing data between threads before processing begins 253
Dividing data recursively 254 ▪ Dividing work by task type 258

8.2 Factors affecting the performance of concurrent code 260

How many processors? 261 ▪ Data contention and cache ping-pong 262 ▪ False sharing 264 ▪ How close is your data? 265 ▪ Oversubscription and excessive task switching 266

8.3 Designing data structures for multithreaded performance 266

Dividing array elements for complex operations 267 ▪ Data access patterns in other data structures 269

8.4 Additional considerations when designing for concurrency 270

Exception safety in parallel algorithms 271 ▪ Scalability and Amdahl's law 277 ▪ Hiding latency with multiple threads 279
Improving responsiveness with concurrency 280

8.5 Designing concurrent code in practice 282

A parallel implementation of std::for_each 282 ▪ A parallel implementation of std::find 284 ▪ A parallel implementation of std::partial_sum 290

9 ***Advanced thread management 300***

9.1 Thread pools 301

The simplest possible thread pool 301 ▪ Waiting for tasks submitted to a thread pool 303 ▪ Tasks that wait for other tasks 307 ▪ Avoiding contention on the work queue 310 Work stealing 311

9.2 Interrupting threads 315

Launching and interrupting another thread 316 ▪ Detecting that a thread has been interrupted 318 ▪ Interrupting a condition variable wait 318 ▪ Interrupting a wait on std::condition_variable_any 321 ▪ Interrupting other blocking calls 323 ▪ Handling interruptions 324 Interrupting background tasks on application exit 325

10 ***Parallel algorithms 327***

10.1 Parallelizing the standard library algorithms 327

10.2 Execution policies 328

General effects of specifying an execution policy 328 std::execution::sequenced_policy 330 std::execution::parallel_policy 330 std::execution::parallel_unsequenced_policy 331

10.3 The parallel algorithms from the C++ Standard Library 331

Examples of using parallel algorithms 334 Counting visits 336

11 ***Testing and debugging multithreaded applications 339***

11.1 Types of concurrency-related bugs 340

Unwanted blocking 340 ▪ Race conditions 341

11.2 Techniques for locating concurrency-related bugs 342

Reviewing code to locate potential bugs 342 ▪ Locating concurrency-related bugs by testing 344 ▪ Designing for testability 346 ▪ Multithreaded testing techniques 347 Structuring multithreaded test code 350 ▪ Testing the performance of multithreaded code 352

appendix A *Brief reference for some C++11 language features* *354*
appendix B *Brief comparison of concurrency libraries* *382*
appendix C *A message-passing framework and complete ATM example* *384*
appendix D *C++ Thread Library reference* *401*

index *551*