# Table of Contents

**Preface** 1

**Chapter 1: A Brief Introduction to C++** 7

- **Why C++?** 7
  - Zero-cost abstractions 7
    - Programming languages and machine code abstractions 8
    - Abstractions in other languages 10
  - Portability 10
  - Robustness 10
  - C++ of today 11
- **The aim of this book** 11
  - Expected knowledge of the reader 11
- **C++ compared with other languages** 11
  - Competing languages and performance 12
  - Non-performance-related C++ language features 15
    - Value semantics 15
    - Const correctness 17
  - Object ownership and garbage collection in C++ 18
  - Avoiding null objects using C++ references 18
  - Drawbacks of C++ 20
- **Class interfaces and exceptions** 21
  - Strict class interfaces 21
  - Error handling and resource acquisition 22
    - Preserving the valid state 23
    - Resource acquisition 24
    - Exceptions versus error codes 25
- **Libraries used in this book** 25
- **Summary** 26

**Chapter 2: Modern C++ Concepts** 27

- **Automatic type deduction with the auto keyword** 27
  - Using auto in function signatures 27
  - Using auto for variables 28
    - Const reference 29
    - Mutable reference 30
    - Forwarding reference 30
    - Conclusion 30

**The lambda function** 31
Basic syntax of a C++ lambda function 31
The capture block 32
Capture by reference versus capture by value 33
Similarities between a Lambda and a class 34
Initializing variables in capture 35
Mutating lambda member variables 35
Mutating member variables from the compiler's perspective 37
Capture all 37
Assigning C function pointers to lambdas 39
Lambdas and std::function 39
Assigning lambdas to std::functions 40
Implementing a simple Button class with std::function 40
Performance consideration of std::function 42
An std::function cannot be inlined 42
An std::function heap allocates and captures variables 42
Invoking an std::function requires a few more operations than a lambda 42
The polymorphic lambda 43
Creating reusable polymorphic lambdas 44
**Const propagation for pointers** 46
**Move semantics explained** 48
Copy-construction, swap, and move 48
Copy-constructing an object 48
Swapping two objects 49
Move-constructing an object 50
Resource acquisition and the rule of three 50
Implementing the rule of three 51
Constructor 52
Limitations of the rule of three 53
Avoiding copies without move semantics 54
Introducing move semantics 54
Named variables and r-values 56
Accept arguments by move when applicable 57
Default move semantics and the rule of zero 58
Rule of zero in a real code base 59
A note on empty destructors 60
A common pitfall - moving non-resources 61
Applying the && modifier to class member functions 63
**Representing optional values with std::optional** 63
Optional return values 63
Optional member variables 64
Sorting and comparing std::optional 65
**Representing dynamic values with std::any** 66
Performance of std::any 66

**Summary** 67

**Chapter 3: Measuring Performance** 69

**Asymptotic complexity and big O notation** 70

Growth rates 75

Amortized time complexity 76

**What to measure?** 78

Performance properties 80

Performance testing – best practices 81

**Knowing your code and hot spots** 81

Profilers 82

Instrumentation profilers 83

Sampling profilers 85

**Summary** 87

**Chapter 4: Data Structures** 89

**Properties of computer memory** 89

**STL containers** 92

Sequence containers 93

Vector and array 93

Deque 96

List and forward_list 96

The basic_string 97

Associative containers 97

Ordered sets and maps 98

Unordered sets and maps 99

Hash and equals 100

Hash policy 102

Container adaptors 102

Priority queues 103

**Parallel arrays** 105

**Summary** 113

**Chapter 5: A Deeper Look at Iterators** 115

**The iterator concept** 115

Iterator categories 116

Pointer-mimicking syntax 117

Iterators as generators 119

Iterator traits 120

Implementing a function using iterator categories 121

Extending the IntIterator to bidirectional 122

Practical example – iterating floating point values within a range 122

Illustrated usage examples 123

Utility functions 124
How to construct a linear range iterator 125
Iterator usage example 126
Generalizing the iterator pair to a range 127
The make_linear_range convenience function 127
Linear range usage examples 128
**Summary** 129

# Chapter 6: STL Algorithms and Beyond 131

**Using STL algorithms as building blocks** 131
STL algorithm concepts 131
Algorithms operate on iterators 132
Implementing a generic algorithm that can be used with any container 132
Iterators for a range point to the first element and the element after the last 134
Algorithms do not change the size of the container 134
Algorithms with output require allocated data 135
Algorithms use operator== and operator< by default 136
Custom comparator function 137
General-purpose predicates 137
Algorithms require move operators not to throw 138
Algorithms have complexity guarantees 139
Algorithms perform just as well as C library function equivalents 140
STL algorithms versus handcrafted for-loops 140
Readability and future-proofing 141
Real-world code base example 141
Usage examples of STL algorithms versus handcrafted for-loops 142
Example 1 – Unfortunate exceptions and performance problems 143
Example 2 – STL has subtle optimizations even in simple algorithms 145
Sorting only for the data you need to retrieve 147
Use cases 148
Performance evaluation 149
**The future of STL and the ranges library** 150
Limitations of the iterators in STL 150
Introduction to the ranges library 153
Composability and pipeability 155
Actions, views, and algorithms 156
Actions 157
Views 158
Algorithms 160
**Summary** 160

# Chapter 7: Memory Management 161

**Computer memory** 162
The virtual address space 162
Memory pages 162

Thrashing 164

**Process memory** 164

Stack memory 165

Heap memory 168

**Objects in memory** 169

Creating and deleting objects 169

Placement new 170

The new and delete operators 171

Memory alignment 173

Padding 175

**Memory ownership** 177

Handling resources implicitly 178

Containers 180

Smart pointers 180

Unique pointer 180

Shared pointer 181

Weak pointer 182

**Small size optimization** 183

**Custom memory management** 186

Building an arena 187

A custom memory allocator 191

**Summary** 196

**Chapter 8: Metaprogramming and Compile-Time Evaluation** 197

**Introduction to template metaprogramming** 197

Using integers as template parameters 199

How the compiler handles a template function 200

Using static_assert to trigger errors at compile time 201

**Type traits** 201

Type trait categories 201

Using type traits 202

Receiving the type of a variable with decltype 203

Conditionally enable functions based on types with std::enable_if_t 204

Introspecting class members with std::is_detected 206

Usage example of is_detected and enable_if_t combined 207

**The constexpr keyword** 209

Constexpr functions in a runtime context 210

Verify compile-time computation using std::integral_constant 210

The if constexpr statement 211

Comparison with runtime polymorphism 212

Example of generic modulus function using if constexpr 213

Heterogeneous containers 214
Static-sized heterogenous containers 214
The std::tuple container 214
Accessing the members of a tuple 215
Iterating std::tuple 216
Unrolling the tuple 217
Implementing other algorithms for tuples 218
Accessing tuple elements 219
Structured bindings 220
The variadic template parameter pack 221
An example of a function with variadic number of arguments 221
How to construct a variadic parameter pack 222
Dynamic-sized heterogenous containers 223
Using std::any as the base for a dynamic-size heterogenous container 223
The std::variant 225
Visiting variants 226
Heterogenous container of variants 227
Accessing the values in our variant container 228
Global function std::get 229
Real world examples of metaprogramming 229
Example 1 – Reflection 229
Making a class reflect its members 230
C++ libraries which simplifies reflection 230
Using the reflection 230
Evaluating the assembler output of the reflection 231
Conditionally overloading global functions 232
Testing reflection capabilities 234
Example 2 – Creating a generic safe cast function 236
Example 3 – Hash strings at compile time 239
The advantages of compile-time hash sum calculation 240
Implement and verify a compile-time hash function 241
Constructing a PrehashedString class 241
Forcing PrehashedString to only accept compile time string literals 243
Evaluating PrehashedString 243
Evaluating get_bitmap_resource() with PrehashedString 244
Summary 245
Chapter 9: Proxy Objects and Lazy Evaluation 247
An introduction to lazy evaluation and proxy objects 247
Lazy versus eager evaluation 247
Proxy objects 249
Comparing concatenated strings using a proxy 249
Implementing the proxy 250

Performance evaluation 252
The r-value modifier 252
Assigning a concatenated proxy 253
**Postponing an sqrt computation when comparing distances** 253
A simple two-dimensional point class 253
The underlying mathematics 254
Implementing the DistProxy object 256
Expanding DistProxy to something more useful 258
Comparing distances with DistProxy 259
Calculating distances with DistProxy 259
Preventing the misuse of DistProxy 260
Performance evaluation 261
**Creative operator overloading and proxy objects** 261
The pipe operator as an extension method 262
The pipe operator 262
The infix operator 263
Further reading 265
**Summary** 265
**Chapter 10: Concurrency** 267
**Understanding the basics of concurrency** 268
**What makes concurrent programming hard?** 268
**Concurrency and parallelism** 269
Time slicing 270
Shared memory 271
Data races 273
Mutex 274
Deadlock 276
Synchronous and asynchronous tasks 277
**Concurrent programming in C++** 278
The thread support library 278
Threads 278
Thread states 281
Protecting critical sections 282
Avoiding deadlocks 284
Condition variables 285
Returning data and handling errors 287
Tasks 288
Atomic support in C++ 290
Using shared_ptr in a multithreaded environment 291
C++ memory model 294

Instruction reordering 295
Atomics and memory orders 296
**Lock-free programming** 298
Lock-free queue example 299
**Performance guidelines** 301
Avoid contention 301
Avoid blocking operations 302
Number of threads/CPU cores 302
Thread priorities 303
Thread affinity 303
False sharing 304
**Summary** 305

**Chapter 11: Parallel STL** 307

**Importance of parallelism** 308
**Parallel algorithms** 308
Implementing parallel std::transform() 309
Naive implementation 309
Performance evaluation 310
Shortcomings of the naive implementation 311
Divide and conquer 313
Implementation 314
Performance evaluation 314
Implementing parallel std::count_if 316
Implementing parallel std::copy_if 317
Approach one – Use a synchronized write position 318
Inner function 318
Outer function 319
Approach two – Split algorithm into two parts 319
Part one – Copy elements in parallel into the destination range 320
Part two – Move the sparse range sequentially into a continuous range 321
Performance evaluation 322
**Parallel STL** 323
Execution policies 323
Sequenced policy 324
Parallel policy 324
Parallel unsequenced policy 325
Parallel modifications of algorithm 325
std::accumulate and std::reduce 325
std::transform_reduce 327
std::for_each 327
Parallelizing an index-based for-loop 328
Combining std::for_each with linear range 329

Simplifying construction via a wrapper 329
**Executing STL algorithms on the GPU** 330
GPU APIs and parallel operations 330
Programmable GPUs 330
Shader programs 331
STL algorithms and the GPU 331
**Boost Compute** 331
Basic concepts of Boost Compute 332
OpenCL 332
Initializing Boost Compute 332
Transfer a simple transform-reduce algorithm to Boost Compute 333
The algorithm in standard C++ 334
Transforming the algorithm to Boost Compute 334
Adapting the circle struct for use with Boost Compute 334
Converting circle_area_cpu to Boost Compute 335
The BOOST_COMPUTE_FUNCTION macro 336
Implementing the transform-reduction algorithm on the GPU 336
Using predicates with Boost Compute 337
Using a custom kernel in Boost Compute 338
Box filter 339
Implementing the kernel 340
Parallelizing for two dimensions 340
Verify GPU computation on the CPU 342
Summary 342
**Other Books You May Enjoy** 345
**Leave a review - let other readers know what you think** 347
**Index** 349