

Table of Contents

Preface	xv
1. An Overview of Optimization	1
Optimization Is Part of Software Development	2
Optimization Is Effective	3
It's OK to Optimize	3
A Nanosecond Here, a Nanosecond There	6
Summary of Strategies for Optimizing C++ Code	6
Use a Better Compiler, Use Your Compiler Better	7
Use Better Algorithms	8
Use Better Libraries	9
Reduce Memory Allocation and Copying	10
Remove Computation	11
Use Better Data Structures	12
Increase Concurrency	12
Optimize Memory Management	12
Summary	12
2. Computer Behavior Affecting Optimization	15
Lies C++ Believes About Computers	16
The Truth About Computers	17
Memory Is Slow	17
Memory Is Not Accessed in Bytes	18
Some Memory Accesses Are Slower than Others	19
Memory Words Have a Big End and a Little End	20
Memory Has Finite Capacity	20
Instruction Execution Is Slow	21
Making Decisions Is Hard for Computers	22

There Are Multiple Streams of Program Execution	22
Calling into the Operating System Is Expensive	24
C++ Tells Lies Too	24
All Statements Are Not Equally Expensive	24
Statements Are Not Executed in Order	25
Summary	26
3. Measure Performance.....	27
The Optimizing Mindset	28
Performance Must Be Measured	28
Optimizers Are Big Game Hunters	29
The 90/10 Rule	29
Amdahl's Law	31
Perform Experiments	32
Keep a Lab Notebook	34
Measure Baseline Performance and Set Goals	35
You Can Improve Only What You Measure	37
Profile Program Execution	37
Time Long-Running Code	40
"A Little Learning" About Measuring Time	40
Measuring Time with Computers	46
Overcoming Measurement Obstacles	54
Create a Stopwatch Class	58
Time Hot Functions in a Test Harness	62
Estimate Code Cost to Find Hot Code	63
Estimate the Cost of Individual C++ Statements	63
Estimate the Cost of Loops	64
Other Ways to Find Hot Spots	66
Summary	67
4. Optimize String Use: A Case Study.....	69
Why Strings Are a Problem	69
Strings Are Dynamically Allocated	70
Strings Are Values	70
Strings Do a Lot of Copying	71
First Attempt at Optimizing Strings	72
Use Mutating String Operations to Eliminate Temporaries	74
Reduce Reallocation by Reserving Storage	74
Eliminate Copying of String Arguments	75
Eliminate Pointer Dereference Using Iterators	76
Eliminate Copying of Returned String Values	77
Use Character Arrays Instead of Strings	78

Summary of First Optimization Attempt	80
Second Attempt at Optimizing Strings	80
Use a Better Algorithm	80
Use a Better Compiler	82
Use a Better String Library	83
Use a Better Allocator	87
Eliminate String Conversion	88
Conversion from C String to std::string	89
Converting Between Character Encodings	89
Summary	90
5. Optimize Algorithms.....	91
Time Cost of Algorithms	92
Best-Case, Average, and Worst-Case Time Cost	95
Amortized Time Cost	95
Other Costs	96
Toolkit to Optimize Searching and Sorting	96
Efficient Search Algorithms	96
Time Cost of Searching Algorithms	97
All Searches Are Equal When n Is Small	98
Efficient Sort Algorithms	98
Time Cost of Sorting Algorithms	99
Replace Sorts Having Poor Worst-Case Performance	99
Exploit Known Properties of the Input Data	100
Optimization Patterns	100
Precomputation	101
Lazy Computation	102
Batching	102
Caching	103
Specialization	104
Taking Bigger Bites	104
Hinting	105
Optimizing the Expected Path	105
Hashing	105
Double-Checking	105
Summary	106
6. Optimize Dynamically Allocated Variables.....	107
C++ Variables Refresher	108
Storage Duration of Variables	108
Ownership of Variables	111
Value Objects and Entity Objects	112

80	C++ Dynamic Variable API Refresher	113
80	Smart Pointers Automate Ownership of Dynamic Variables	116
80	Dynamic Variables Have Runtime Cost	118
82	Reduce Use of Dynamic Variables	119
83	Create Class Instances Staticly	119
87	Use Static Data Structures	121
88	Use <code>std::make_shared</code> Instead of <code>new</code>	124
88	Don't Share Ownership Unnecessarily	125
88	Use a "Master Pointer" to Own Dynamic Variables	126
90	Reduce Reallocation of Dynamic Variables	127
90	Preallocate Dynamic Variables to Prevent Reallocation	127
91	Create Dynamic Variables Outside of Loops	128
92	Eliminate Unneeded Copying	129
92	Disable Unwanted Copying in the Class Definition	130
92	Eliminate Copying on Function Call	131
92	Eliminate Copying on Function Return	132
92	Copy Free Libraries	134
92	Implement the "Copy on Write" Idiom	136
97	Slice Data Structures	137
98	Implement Move Semantics	137
98	Nonstandard Copy Semantics: A Painful Hack	138
99	<code>std::swap()</code> : The Poor Man's Move Semantics	138
99	Shared Ownership of Entities	139
100	The Moving Parts of Move Semantics	140
100	Update Code to Use Move Semantics	141
101	Subtleties of Move Semantics	142
102	Flatten Data Structures	145
102	Summary	146
103	7. Optimize Hot Statements	147
104	Remove Code from Loops	148
102	Cache the Loop End Value	149
102	Use More Efficient Loop Statements	149
102	Count Down Instead of Up	150
102	Remove Invariant Code from Loops	151
102	Remove Unneeded Function Calls from Loops	152
107	Remove Hidden Function Calls from Loops	155
107	Remove Expensive, Slow-Changing Calls from Loops	156
108	Push Loops Down into Functions to Reduce Call Overhead	157
108	Do Some Actions Less Frequently	158
111	What About Everything Else?	160
112	Remove Code from Functions	160

Cost of Function Calls	161
Declare Brief Functions Inline	165
Define Functions Before First Use	165
Eliminate Unused Polymorphism	165
Discard Unused Interfaces	166
Select Implementation at Compile Time with Templates	170
Eliminate Uses of the PIMPL Idiom	171
Eliminate Calls into DLLs	173
Use Static Member Functions Instead of Member Functions	173
Move Virtual Destructor to Base Class	174
Optimize Expressions	174
Simplify Expressions	175
Group Constants Together	176
Use Less-Expensive Operators	177
Use Integer Arithmetic Instead of Floating Arithmetic	177
Double May Be Faster than Float	179
Replace Iterative Computations with Closed Forms	180
Optimize Control Flow Idioms	182
Use switch Instead of if-elseif-else	182
Use Virtual Functions Instead of switch or if	182
Use No-Cost Exception Handling	183
Summary	185
8. Use Better Libraries.....	187
Optimize Standard Library Use	187
Philosophy of the C++ Standard Library	188
Issues in Use of the C++ Standard Library	188
Optimize Existing Libraries	191
Change as Little as Possible	191
Add Functions Rather than Change Functionality	192
Design Optimized Libraries	192
Code in Haste, Repent at Leisure	193
Parsimony Is a Virtue in Library Design	194
Make Memory Allocation Decisions Outside the Library	194
When in Doubt, Code Libraries for Speed	195
Functions Are Easier to Optimize than Frameworks	195
Flatten Inheritance Hierarchies	196
Flatten Calling Chains	196
Flatten Layered Designs	196
Avoid Dynamic Lookup	198
Beware of 'God Functions'	199
Summary	200

9. Optimize Searching and Sorting.....	201
Key/Value Tables Using <code>std::map</code> and <code>std::string</code>	202
Toolkit to Improve Search Performance	203
Make a Baseline Measurement	204
Identify the Activity to Be Optimized	204
Decompose the Activity to Be Optimized	205
Change or Replace Algorithms and Data Structures	206
Using the Optimization Process on Custom Abstractions	208
Optimize Search Using <code>std::map</code>	208
Use Fixed-Size Character Array Keys with <code>std::map</code>	208
Use C-Style String Keys with <code>std::map</code>	210
Using Map's Cousin <code>std::set</code> When the Key Is in the Value	212
Optimize Search Using the <code><algorithm></code> Header	213
Key/Value Table for Search in Sequence Containers	214
<code>std::find()</code> : Obvious Name, $O(n)$ Time Cost	215
<code>std::binary_search()</code> : Does Not Return Values	216
Binary Search Using <code>std::equal_range()</code>	216
Binary Search Using <code>std::lower_bound()</code>	217
Handcoded Binary Search	218
Handcoded Binary Search using <code>strcmp()</code>	219
Optimize Search in Hashed Key/Value Tables	220
Hashing with a <code>std::unordered_map</code>	221
Hashing with Fixed Character Array Keys	221
Hashing with Null-Terminated String Keys.....	222
Hashing with a Custom Hash Table	224
Stepanov's Abstraction Penalty	225
Optimize Sorting with the C++ Standard Library	226
Summary	228
10. Optimize Data Structures.....	229
Get to Know the Standard Library Containers	229
Sequence Containers	230
Associative Containers	230
Experimenting with the Standard Library Containers	231
<code>std::vector</code> and <code>std::string</code>	236
Performance Consequences of Reallocation	237
Inserting and Deleting in <code>std::vector</code>	238
Iterating in <code>std::vector</code>	240
Sorting <code>std::vector</code>	241
Lookup with <code>std::vector</code>	241
<code>std::deque</code>	242
Inserting and Deleting in <code>std::deque</code>	243

Iterating in <code>std::deque</code>	245
Sorting <code>std::deque</code>	245
Lookup with <code>std::deque</code>	245
<code>std::list</code>	245
Inserting and Deleting in <code>std::list</code>	247
Iterating in <code>std::list</code>	248
Sorting <code>std::list</code>	248
Lookup with <code>std::list</code>	249
<code>std::forward_list</code>	249
Inserting and Deleting in <code>std::forward_list</code>	250
Iterating in <code>std::forward_list</code>	251
Sorting <code>std::forward_list</code>	251
Lookup in <code>std::forward_list</code>	251
<code>std::map</code> and <code>std::multimap</code>	251
Inserting and Deleting in <code>std::map</code>	252
Iterating in <code>std::map</code>	255
Sorting <code>std::map</code>	255
Lookup with <code>std::map</code>	255
<code>std::set</code> and <code>std::multiset</code>	255
<code>std::unordered_map</code> and <code>std::unordered_multimap</code>	256
Inserting and Deleting in <code>std::unordered_map</code>	260
Iterating in <code>std::unordered_map</code>	260
Lookup with <code>std::unordered_map</code>	261
Other Data Structures	261
Summary	263
11. Optimize I/O.....	265
A Recipe for Reading Files	265
Create a Parsimonious Function Signature	267
Shorten Calling Chains	269
Reduce Reallocation	269
Take Bigger Bites—Use a Bigger Input Buffer	272
Take Bigger Bites—Read a Line at a Time	272
Shorten Calling Chains Again	274
Things That Didn't Help.....	275
Writing Files	276
Reading from <code>std::cin</code> and Writing to <code>std::cout</code>	277
Summary	278
12. Optimize Concurrency.....	279
Concurrency Refresher	280
A Walk Through the Concurrency Zoo	281

Interleaved Execution	285
Sequential Consistency	286
Races	287
Synchronization	288
Atomicity	289
C++ Concurrency Facilities Refresher	291
Threads	291
Promises and Futures	292
Asynchronous Tasks	295
Mutexes	296
Locks	297
Condition Variables	298
Atomic Operations on Shared Variables	301
On Deck: Future C++ Concurrency Features	304
Optimize Threaded C++ Programs	305
Prefer <code>std::async</code> to <code>std::thread</code>	306
Create as Many Runnable Threads as Cores	308
Implement a Task Queue and Thread Pool	309
Perform I/O in a Separate Thread	310
Program Without Synchronization	310
Remove Code from Startup and Shutdown	313
Make Synchronization More Efficient	314
Reduce the Scope of Critical Sections	314
Limit the Number of Concurrent Threads	315
Avoid the Thundering Herd	316
Avoid Lock Convoys	317
Reduce Contention	317
Don't Busy-Wait on a Single-Core System	319
Don't Wait Forever	319
Rolling Your Own Mutex May Be Ineffective	319
Limit Producer Output Queue Length	320
Concurrency Libraries	320
Summary	322
13. Optimize Memory Management	323
C++ Memory Management API Refresher	324
The Life Cycle of Dynamic Variables	324
Memory Management Functions Allocate and Free Memory	325
New-Expressions Construct Dynamic Variables	328
Delete-Expressions Dispose of Dynamic Variables	331
Explicit Destructor Calls Destroy Dynamic Variables	332
High-Performance Memory Managers	333

Provide Class-Specific Memory Managers 335

- Fixed-Size-Block Memory Manager 336
- Block Arena 338
- Adding a Class-Specific operator new() 340
- Performance of the Fixed-Block Memory Manager 342
- Variations on the Fixed-Block Memory Manager 342
- Non-Thread Safe Memory Managers Are Efficient 343

Provide Custom Standard Library Allocators 343

- Minimal C++11 Allocator 346
- Additional Definitions for C++98 Allocator 347
- A Fixed-Block Allocator 352
- A Fixed-Block Allocator for Strings 354

Summary 355

Index..... 357

Hi, My name is Kurt, and I'm a code-aholic.

Google, Facebook, Apple, or anywhere else famous. But beyond a few short vacations, I have written code every day of that time. I have spent the last 20 years almost exclusively writing C++ and talking to other very bright developers about C++. This is my qualification to write a book about optimizing C++ code. I have also written a lot of English prose, including specifications, comments, manuals, notes, and blog posts (<http://oldhandsblog.blogspot.com>). It has amazed me from time to time that only half of the bright, competent developers I have worked with can string two grammatical English sentences together.

One of my favorite quotes comes by way of a letter from Sir Isaac Newton, in which he writes, "If I have seen farther, it is by standing on the shoulders of giants." I too have stood on the shoulders of giants, and particularly have read their book: elegant little books, like Brian Kernighan and Dennis Ritchie's *The C Programming Language*; smart, ahead-of-the-curve books, like Scott Meyers's *Effective C++* series; challenging, mind-expanding books, like Andrei Alexandrescu's *Modern C++ Design*; careful, precise books, like Bjarne Stroustrup and Margaret Ellis's *The Annotated C++ Reference Manual*. For most of my career, it never crossed my mind that I might someday write a book. Then one day, quite suddenly, I found I needed to write this one.

So why write a book about performance tuning in C++?

At the dawn of the 21st century, C++ was under assault. Fans of C pointed to C++ programs whose performance was inferior to supposedly equivalent code written in C. Famous corporations with big marketing budgets touted proprietary object-oriented languages, claiming C++ was too hard to use, and that their tools were the future. Universities settled on Java for teaching because it came with a free toolchain. As a result of all this buzz, big companies made big-money bets on coding websites and operating systems in Java or C# or PHP. C++ seemed to be on the way. It was an uncomfortable time for anyone who believed C++ was a powerful, useful tool.