

Contents

1	You Write Software; You have Bugs	1
2	A Systematic Approach to Debugging	5
2.1	Why Follow a Structured Process?	5
2.2	Making the Most of Your Opportunities	5
2.3	13 Golden Rules	7
2.3.1	Understand the Requirements	8
2.3.2	Make it Fail	8
2.3.3	Simplify the Test Case	9
2.3.4	Read the Right Error Message	9
2.3.5	Check the Plug	9
2.3.6	Separate Facts from Interpretation	10
2.3.7	Divide and Conquer	10
2.3.8	Match the Tool to the Bug	12
2.3.9	One Change at a Time	12
2.3.10	Keep an Audit Trail	12
2.3.11	Get a Fresh View	13
2.3.12	If You Didn't Fix it, it Ain't Fixed	13
2.3.13	Cover your Bugfix with a Regression Test	13
2.4	Build a Good Toolkit	14
2.4.1	Your Workshop	15
2.4.2	Running Tests Every Day Keeps the Bugs at Bay	15
2.5	Know Your Enemy – Meet the Bug Family	17
2.5.1	The Common Bug	17
2.5.2	Sporadic Bugs	18
2.5.3	Heisenbugs	18
2.5.4	Bugs Hiding Behind Bugs	19
2.5.5	Secret Bugs – Debugging and Confidentiality	20
2.5.6	Further Reading	21

3	Getting to the Root – Source Code Debuggers	23
3.1	Visualizing Program Behavior	23
3.2	Prepare a Simple Predictable Example	24
3.3	Get the Debugger to Run with Your Program	24
3.4	Learn to do a Stack Trace on a Program Crash	27
3.5	Learn to Use Breakpoints	28
3.6	Learn to Navigate Through the Program	28
3.7	Learn to Inspect Data: Variables and Expressions	29
3.8	A Debug Session on a Simple Example	30
4	Fixing Memory Problems	33
4.1	Memory Management in C/C++ – Powerful but Dangerous	33
4.1.1	Memory Leaks	34
4.1.2	Incorrect Use of Memory Management	34
4.1.3	Buffer Overruns	34
4.1.4	Uninitialized Memory Bugs	34
4.2	Memory Debuggers to the Rescue	35
4.3	Example 1: Detecting Memory Access Errors	36
4.3.1	Detecting an Invalid Write Access	36
4.3.2	Detecting Uninitialized Memory Reads	37
4.3.3	Detecting Memory Leaks	38
4.4	Example 2: Broken Calls to Memory Allocation/Deallocation	38
4.5	Combining Memory and Source Code Debuggers	40
4.6	Cutting Down the Noise – Suppressing Errors	40
4.7	When to Use a Memory Debugger	41
4.8	Restrictions	42
4.8.1	Prepare Test Cases with Good Code Coverage	42
4.8.2	Provide Additional Computer Resources	42
4.8.3	Multi-Threading May Not be Supported	42
4.8.4	Support for Non-standard Memory Handlers	42
5	Profiling Memory Use	45
5.1	Basic Strategy – The First Steps	45
5.2	Example 1: Allocating Arrays	46
5.3	Step 1: Look for Leaks	46
5.4	Step 2: Set Your Expectations	47
5.5	Step 3: Measure Memory Consumption	47
5.5.1	Use Multiple Inputs	48
5.5.2	Stopping the Program at Regular Intervals	48
5.5.3	Measuring Memory Consumption with Simple Tools	49
5.5.4	Use <code>top</code>	49
5.5.5	Use the Windows Task Manager	50
5.5.6	Select Relevant Input Values for <code>testmalloc</code>	51
5.5.7	Determine how Memory is Deallocated on Your Machine	51
5.5.8	Use a Memory Profiler	53

5.6	Step 4: Identifying Greedy Data Structures	54
5.6.1	Instrumenting Data Structures	55
5.7	Putting it Together – The <code>genindex</code> Example	55
5.7.1	Check that There are No Major Leaks	56
5.7.2	Estimate the Expected Memory Use	56
5.7.3	Measure Memory Consumption	57
5.7.4	Find the Data Structures that Consume Memory	57
6	Solving Performance Problems	63
6.1	Finding Performance Bugs – A Step-by-Step Approach	63
6.1.1	Do an Upfront Analysis	64
6.1.2	Use a Simple Method of Measuring Time	64
6.1.3	Create a Test Case	65
6.1.4	Make the Test Case Reproducible	65
6.1.5	Check the Program for Correctness	66
6.1.6	Make the Test Case Scalable	66
6.1.7	Isolate the Test Case from Side Effects	67
6.1.8	Measurement with <code>time</code> can have Errors and Variations ..	68
6.1.9	Select a Test Case that Exposes the Runtime Bottleneck ..	68
6.1.10	The Difference Between Algorithm and Implementation ..	70
6.2	Using Profiling Tools	72
6.2.1	Do Not Write Your Own Profiler	72
6.2.2	How Profilers Work	73
6.2.3	Familiarize Yourself with <code>gprof</code>	74
6.2.4	Familiarize Yourself with <code>Quantify</code>	79
6.2.5	Familiarize Yourself with <code>Callgrind</code>	81
6.2.6	Familiarize Yourself with <code>VTune</code>	82
6.3	Analyzing I/O Performance	84
6.3.1	Do a Sanity Check of Your Measurements	85
7	Debugging Parallel Programs	87
7.1	Writing Parallel Programs in C/C++	87
7.2	Debugging Race Conditions	88
7.2.1	Using Basic Debugger Capabilities to Find Race Conditions	89
7.2.2	Using Log Files to Localize Race Conditions	91
7.3	Debugging Deadlocks	93
7.3.1	How to Determine What the Current Thread is Executing ..	94
7.3.2	Analyzing the Threads of the Program	95
7.4	Familiarize Yourself with Threading Analysis Tools	96
7.5	Asynchronous Events and Interrupt Handlers	98

8	Finding Environment and Compiler Problems	101
8.1	Environment Changes – Where Problems Begin	101
8.1.1	Environment Variables	101
8.1.2	Local Installation Dependencies	102
8.1.3	Current Working Directory Dependency	102
8.1.4	Process ID Dependency	102
8.2	How else to See what a Program is Doing	103
8.2.1	Viewing Processes with <code>top</code>	103
8.2.2	Finding Multiple Processes of an Application with <code>ps</code>	103
8.2.3	Using <code>/proc/<pid></code> to Access a Process	104
8.2.4	Use <code>strace</code> to Trace Calls to the OS	104
8.3	Compilers and Debuggers have Bugs too	106
8.3.1	Compiler Bugs	106
8.3.2	Debugger and Compiler Compatibility Problems	107
9	Dealing with Linking Problems	109
9.1	How a Linker Works	109
9.2	Building and Linking Objects	110
9.3	Resolving Undefined Symbols	111
9.3.1	Missing Linker Arguments	111
9.3.2	Searching for Missing Symbols	112
9.3.3	Linking Order Issues	113
9.3.4	C++ Symbols and Name Mangling	114
9.3.5	Demangling of Symbols	115
9.3.6	Linking C and C++ Code	115
9.4	Symbols with Multiple Definitions	116
9.5	Symbol Clashes	117
9.6	Identifying Compiler and Linker Version Mismatches	118
9.6.1	Mismatching System Libraries	119
9.6.2	Mismatching Object Files	119
9.6.3	Runtime Crashes	120
9.6.4	Determining the Compiler Version	120
9.7	Solving Dynamic Linking Issues	122
9.7.1	Linking or Loading DLLs	122
9.7.2	DLL Not Found	124
9.7.3	Analyzing Loader Issues	125
9.7.4	Setting Breakpoints in DLLs	126
9.7.5	Provide Error Messages for DLL Issues	127
10	Advanced Debugging	129
10.1	Setting Breakpoints in C++ Functions, Methods, and Operators	129
10.2	Setting Breakpoints in Templated Functions and C++ Classes	131
10.3	Stepping in C++ Methods	133
10.3.1	Stepping into Implicit Functions	134
10.3.2	Stepping Implicit Functions with the Step-out Command	135

10.3.3	Skipping Implicit Functions with a Temporary Breakpoint	136
10.3.4	Returning from Implicit Function Calls	136
10.4	Conditional Breakpoints and Breakpoint Commands	137
10.5	Debugging Static Constructor/Destructor Problems	140
10.5.1	Bugs Due to Order-Dependence of Static Initializers	140
10.5.2	Recognizing the Stack Trace of Static Initializers	141
10.5.3	Attaching the Debugger Before Static Initialization	142
10.6	Using Watchpoints	143
10.7	Catching Signals	144
10.8	Catching Exceptions	147
10.9	Reading Stack Traces	148
10.9.1	Stack Trace of Source Code Compiled with Debug Information	148
10.9.2	Stack Trace of Source Code Compiled Without Debug Information	149
10.9.3	Frames Without Any Debug Information	149
10.9.4	Real-Life Stack Traces	150
10.9.5	Mangled Function Names	151
10.9.6	Broken Stack Traces	151
10.9.7	Core Dumps	152
10.10	Manipulating a Running Program	153
10.10.1	Changing a Variable	156
10.10.2	Calling Functions	156
10.10.3	Changing the Return Value of a Function	157
10.10.4	Aborting Function Calls	157
10.10.5	Skipping or Repeating Individual Statements	158
10.10.6	Printing and Modifying Memory Content	159
10.11	Debugging Without Debug Information	161
10.11.1	Reading Function Arguments From the Stack	163
10.11.2	Reading Local/Global Variables, User-Defined Data Types	165
10.11.3	Finding the Approximate Statement in the Source Code	165
10.11.4	Stepping Through Assembly Code	166
11	Writing Debuggable Code	169
11.1	Why Comments Count	169
11.1.1	Comments on Function Signatures	170
11.1.2	Comments on Workarounds	171
11.1.3	Comments in Case of Doubt	171
11.2	Adopting a Consistent Programming Style	171
11.2.1	Choose Names Carefully	171
11.2.2	Avoid Insanely Clever Constructs	172
11.2.3	Spread Out Your Code	172
11.2.4	Use Temporary Variables for Complex Expressions	172
11.3	Avoiding Preprocessor Macros	173
11.3.1	Use Constants or Enums Instead of Macros	173

11.3.2	Use Functions Instead of Preprocessor Macros	175
11.3.3	Debug the Preprocessor Output	176
11.3.4	Consider Using More Powerful Preprocessors	177
11.4	Providing Additional Debugging Functions	179
11.4.1	Displaying User-Defined Data Types	179
11.4.2	Self-Checking Code	180
11.4.3	Debug Helpers for Operators	181
11.5	Prepare for Post-Processing	181
11.5.1	Generate Log Files	181
12	How Static Checking Can Help	183
12.1	Using Compilers as Debugging Tools	183
12.1.1	Do not Assume Warnings to be Harmless	184
12.1.2	Use Multiple Compilers to Check the Code	186
12.2	Using <code>lint</code>	186
12.3	Using Static Analysis Tools	187
12.3.1	Familiarize Yourself with a Static Checker	187
12.3.2	Reduce Static Checker Errors to (Almost) Zero	189
12.3.3	Rerun All Test Cases After a Code Cleanup	190
12.4	Beyond Static Analysis	190
13	Summary	191
A	Debugger Commands	193
B	Access to Tools	195
B.1	IDEs, Compilers, Build Tools	195
B.1.1	Microsoft Visual Studio	195
B.1.2	Eclipse	196
B.1.3	GCC	196
B.1.4	GNU Make	196
B.2	Debuggers	196
B.2.1	<code>dbx</code>	196
B.2.2	<code>DDD</code>	197
B.2.3	<code>GDB</code>	197
B.2.4	ARM RealView	197
B.2.5	TotalView Debugger	197
B.2.6	Lauterbach TRACE32	197
B.3	Environments	198
B.3.1	Cygwin	198
B.3.2	VMware	198
B.4	Memory Debuggers	198
B.4.1	Purify	198
B.4.2	Valgrind	199
B.4.3	KCachegrind	199
B.4.4	Insure++	199

B.4.5	BoundsChecker	200
B.5	Profilers	200
B.5.1	gprof	200
B.5.2	Quantify	200
B.5.3	Intel VTune	200
B.5.4	AQtime	201
B.5.5	mpatrol	201
B.6	Static Checkers	201
B.6.1	Coverity	201
B.6.2	Lint	201
B.6.3	Splint	202
B.6.4	/analyze option in Visual Studio Enterprise Versions ..	202
B.6.5	Klocwork	202
B.6.6	Fortify	202
B.6.7	PC-lint/FlexeLint	203
B.6.8	QA C++	203
B.6.9	Codecheck	203
B.6.10	Axivion Bauhaus Suite	203
B.6.11	C++ SoftBench CodeAdvisor	203
B.6.12	Parasoft C++test	204
B.6.13	LDRA tool suite	204
B.6.14	Understand C++	204
B.7	Tools for Parallel Programming	204
B.7.1	Posix Threads	204
B.7.2	OpenMP	204
B.7.3	Intel TBB	205
B.7.4	MPI	205
B.7.5	MapReduce	205
B.7.6	Intel Threading Analysis Tools	205
B.8	Miscellaneous Tools	206
B.8.1	GNU Binutils	206
B.8.2	m4	206
B.8.3	ps	206
B.8.4	strace/truss	207
B.8.5	top	207
B.8.6	VNC	207
B.8.7	WebEx	207
C	Source Code	209
C.1	testmalloc.c	209
C.2	genindex.c	210
C.3	isort.c	214
C.4	filebug.c	216
	References	217
Index		219