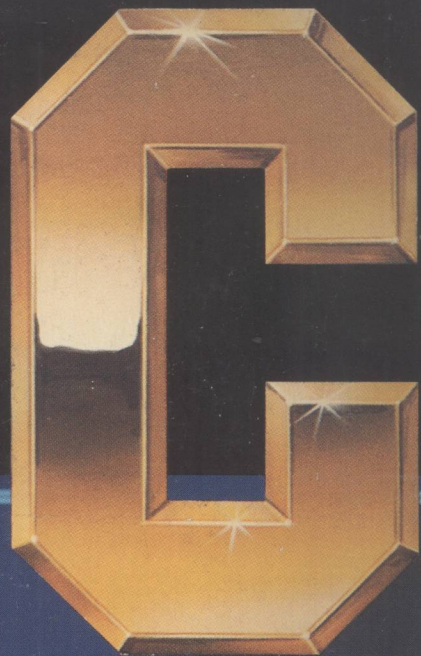


DEBUGGING



que

TP306
W262

8860995

Debugging C

Robert Ward

Que™ Corporation
Indianapolis, Indiana



E8860995

Debugging C. Copyright © 1986 by Que Corporation.

All rights reserved. Printed in the United States of America. No part of this book may be used or reproduced in any form or by any means, or stored in a database or retrieval system, without prior written permission of the publisher except in the case of brief quotations embodied in critical articles and reviews. Making copies of any part of this book for any purpose other than your own personal use is a violation of United States copyright laws. For information, address Que Corporation, P.O. Box 50507, Indianapolis, IN 46250.

Library of Congress Catalog No.: LC 86-61152

ISBN 0-88022-261-1

This book is sold *as is*, without warranty of any kind, either express or implied, respecting the contents of this book, including but not limited to implied warranties for the book's quality, performance, merchantability, or fitness for any particular purpose. Neither Que Corporation nor its dealers or distributors shall be liable to the purchaser or any other person or entity with respect to any liability, loss, or damage caused or alleged to be caused directly or indirectly by this book.

90 89 88 87 86 8 7 6 5 4 3 2 1

Interpretation of the printing code: the rightmost double-digit number is the year of the book's printing; the rightmost single-digit number, the number of the book's printing. For example, a printing code of 87-4 shows that the fourth printing of this book occurred in 1987.

About the Author



Robert Ward

A digital design engineer, Robert Ward develops microprocessor-based communications equipment for an international marketer of computer accessories. He has developed programs ranging from dedicated real-time control programs for single-chip microcontrollers to small compilers for large UNIX hosts to massive vertical applications for CP/M hosts. Ward teaches computer science at McPherson College and is president of Dedicated Micro-Systems, Inc.

His quest for solutions to the challenges of programming in C on a personal computer led to his role as international coordinator of the C User's Group. Ward's interests include programming languages, computer architecture, and artificial intelligence.

Product Director
Chris DeVoney

Editorial Director
David F. Noble, Ph.D.

Managing Editor
Gregory Croy

Editors
Kathleen A. Johanningsmeier
Gail S. Burlakoff

Technical Editor
Paul Wilt

Production Foreman
Dennis Sheehan

Production
Joe Ramon
Mae Louise Shinault
Peter Tocco
Lynne Tone

Composed in Megaron and Que Digital
by Que Corporation

Cover designed by
Listenberger Design Associates

Acknowledgments

There are special people who have contributed significantly to this book. I offer my heartfelt thanks to the following persons:

Chris DeVoney and the technical staff at Que, who have enlightened me about several of my programming parochialisms.

Que's editors, who enhanced the consistency and readability of the manuscripts and brought the entire work to press in a remarkably short time.

David Raanan, of AT&T Information Systems, who reviewed the discussion of `sdb` in Chapter 9.

The good people at C.L. Publications, who created a forum where the main thesis of this book could be tested.

Jack Purdum of EcoSoft, who offered the special encouragement and advocacy that permitted me to take this work from an idea to a published book.

Trademark Acknowledgments

Que Corporation has made every attempt to supply trademark information about company names, products, and services mentioned in this book. Trademarks indicated below were derived from various sources. Que Corporation cannot attest to the accuracy of this information.

Ashton-Tate is a registered trademark of Ashton-Tate Company.

CodeView is a trademark of Microsoft Corporation.

CP/M is a registered trademark of Digital Research, Inc.

C-terp is a trademark of Gimpel Software.

dBASE II is a registered trademark of Ashton-Tate Company.

DDT is a trademark of Digital Research, Inc.

Eco-C88 and Eco-C Compiler are trademarks of Ecosoft, Inc.

Interactive-C is a trademark of IMPACC Associates, Inc.

IBM is a registered trademark of International Business Machines Corporation.

Lattice is a registered trademark of Lattice, Inc.

Microsoft and MS-DOS are registered trademarks of Microsoft Corporation.

PC-Lint is a trademark of Gimpel Software.

UNIX is a registered trademark of AT&T. AT&T is a registered trademark of American Telephone & Telegraph.

ZSID is a trademark of Digital Research, Inc.

Z80 is a registered trademark of Zilog, Inc.

More Computer Knowledge from Que

LOTUS SOFTWARE TITLES	1-2-3 for Business	\$18.95
	1-2-3 Business Formula Handbook	19.95
	1-2-3 Command Language	19.95
	1-2-3 Financial Macros	19.95
	1-2-3 Macro Library, 2nd Edition	19.95
	1-2-3 Tips, Tricks, and Traps, 2nd Edition	19.95
	Using 1-2-3, 2nd Edition	21.95
	Using 1-2-3 Workbook and Disk, 2nd Edition	29.95
	Using Symphony	23.95
	Symphony: Advanced Topics	19.95
	Symphony Macros and the Command Language	22.95
	Symphony Tips, Tricks, and Traps	21.95
	dBASE III Plus Applications Library	19.95
	dBASE III Plus Handbook, 2nd Edition	19.95
DATABASE TITLES	dBASE III Advanced Programming	22.95
	R:base 5000 Techniques and Applications	19.95
	R:base 5000 User's Guide	19.95
	Reflex Tips, Tricks, and Traps	19.95
	Using Reflex	19.95
	Using Paradox	19.95
	Using Q & A	19.95
	Excel Macro Library	19.95
	Multiplan Models for Business	15.95
	Using AppleWorks	16.95
	Using Dollars and Sense	16.95
	Using Enable	19.95
	Using Excel	19.95
	Using Javelin	21.95
APPLICATIONS SOFTWARE TITLES	Using Smart	22.95
	Using DisplayWrite	18.95
	Using MicroSoft Word	16.95
	Using MultiMate	18.95
	Using the PFS Family: FILE, WRITE, GRAPH, REPORT	14.95
	Using WordPerfect, Revised Edition	18.95
	Using WordStar 2000	17.95
	Networking IBM PCs: A Practical Guide, 2nd Edition	19.95
	Using PC DOS	21.95
	Amiga Programming Guide	18.95
	CP/M Programmer's Encyclopedia	19.95
	Managing Your Hard Disk	19.95
	MS-DOS User's Guide	19.95
	Using NetWare	24.95
WORD-PROCESSING TITLES	Advanced C: Techniques and Applications	21.95
	C Programmer's Library	21.95
	C Programming Guide, 2nd Edition	19.95
	C Self-Study Guide	16.95
	Common C Functions	18.95
	Debugging C	19.95
	Turbo Pascal for BASIC Programmers	16.95
	Turbo Pascal Program Library	19.95
	Understanding UNIX: A Conceptual Guide	19.95
	Understanding XENIX: A Conceptual Guide	19.95
	IBM TITLES	
	COMPUTER SYSTEMS TITLES	
	PROGRAMMING AND TECHNICAL TITLES	

Que Order Line: 1-800-428-5331

All prices subject to change without notice.

Table of Contents

Introduction

Why a Book on Debugging?	1
Debugging Then and Debugging Now	1
Debugging = Working Programs	2
Debugging C is Demanding	3
Who Should Read This Book?	4
Chapter Survey	5

1 Foundations

Debugging and Scientific Method	8
The Debugging Process	9
Phase One: Testing	10
Phase Two: Stabilization	10
Phase Three: Localization	11
Phase Four: Correction	12
Principles of Proximity	12
Lexical Proximity	13
Temporal Proximity	14
Referential Proximity	15
Debugging Tools and Techniques.....	16
RETURN	17

2 Program Testing

Recognizing Bugs	19
Lexical Errors	20
Syntactic Errors	20
Execution Errors	21
Intent Errors	23
Other Yardsticks	24
Some Bugs Stand Still	25
Some Bugs Are Input Dependent	25
Some Bugs Are Code Sensitive	26
Some Bugs Are Environment Sensitive	26

Before Attempting To Localize, Stabilize	27
Testing Strategies	27
How Testing Works	28
Top-Down Testing	29
Bottom-Up Testing	33
A Mixed Strategy	35
Selecting Test Data	37
Keeping Adequate Records	39
Hunting versus Verifying	41
RETURN	42

3 Localizing Compile-Time Errors

Compiling: Process and Components	45
Syntax Errors: Some All-Purpose Advice	49
Forcing More Useful Error Messages	50
Using Curly Brace Checkers	59
Using Lint To Analyze Syntax	62
Isolating Bugs with Tiny Test Programs	66
Using the Preprocessor as a Stand-Alone Program	68
Understanding C Syntax	69
Using Hand Parsing	70
Using a Syntax-Directed Editor	70
RETURN	71

4 Conventional Trace Methods

Control-Flow Tracing	74
Instrumenting If Statements	75
Tracing Control Values	75
Instrumenting Loops	76
Instrumenting Breaks and Continues	77
Instrumenting Case Statements	78
Methodical Instrumentation, Not Mystical Insight	81
Instrumenting Functions	82
Data-Flow Tracing	83
Selecting Variables	84
Using Snapshots	84
Tracing Local Variables	87
RETURN	92

5 Managing Trace Facilities

Controlling Trace Output	94
Making Control More Convenient	95
Setting the Control Variable	96
Temporal Switches	96
Hard-Wired Changes	97
Counting Mile Posts	97
Function-by-Function	99
Granularity	102
Practical Variations	104
To the Reader	104
Managing Source Code	105
RETURN	107

6 Why Is Debugging C Difficult?

Strong Types and Error Detection	110
Orthogonality and Error Detection	114
General Impact of Structural Differences	115
The Virtual Machine and Our Expectations	116
Bugs that Attack the Virtual Machine	118
Pointer Bugs and the Stack	118
Out-of-Range Subscripts	122
Variations	123
Uninitialized Pointers	124
Pointers that Write on Code	128
RETURN	130

7 Stabilizing Pointer Bugs

The Importance of Unallocated Memory	134
The Benefits of Initialized Memory	134
Initializing Unallocated Memory	
with a Debug Tool	135
Initializing Memory Using DDT	136
Initializing Memory Using MS-DOS's DEBUG	137
Generating a Load Map	138
8086 Address Notation	141
Identifying Globals	143
Using DEBUG	144
Exploring with DEBUG	146
Words of Caution	147

Building a Memory-Initialization Function	147
Adding a Special Initialization Program to the Invocation Sequence	149
MS-DOS Initialization Programs	149
Special Loaders	152
Initializing Local Variables	153
Accessing the Stack	153
Automating Local Traces	155
RETURN	159

8 Special Trace Techniques

Monitoring the Virtual Machine	161
Checksums on Code Space	162
Checksums under CP/M	164
Problems with CP/M Checksums	166
Checksums under MS-DOS	166
Managing Checksums	167
Stack Walk-Back	167
A Sample Walk-Back	169
A Pointer Bug that Writes on the Stack	173
Interpreting the Stack-Trace Detail	179
Return Addresses	180
The Stack Address and the Frame Pointer	183
Machine-Level Tracing	183
Trace Preparation	184
Displaying the Code for a Specific Function	186
Tracing Execution	191
Monitoring Local Variables with DEBUG	195
RETURN	208

9 Source-Level Debuggers

<i>sdb</i> : UNIX's Symbolic Debugging Program	212
Using <i>sdb</i>	213
Command Format	213
Function and Variable Locators	214
Other Locators	216
Commands that Display Code	216
Commands that Display Variables	222
Commands that Manipulate Breakpoints	229
Commands that Control Execution	235
Direct Function Evaluation	236

The Monitor Command	237
The Stack Traceback Command	237
<i>sdb</i> Application: Postmortem	237
<i>sdb</i> Application: Locating Pointer	
Bugs that Write on Globals	239
<i>sdb</i> Application: Locating Pointer Bugs that	
Write on Locals	240
<i>sdb</i> Application: Locating Pointer Bugs	
that Destroy Return Addresses	241
Testing Modules	250
Caution	251
<i>sdb</i> Weaknesses	251
Other UNIX Debugging Aids	251
CodeView: Microsoft's Source-Level Debugger	252
User Interface	253
Expression Evaluation	256
Watch Windows	256
CodeView Weaknesses	259
RETURN	260

10 Interpreters and Integrated Environments

Edit-Compile-Link Costs	263
Interpretation versus Compilation	265
Controlling Debugging Facilities	265
An Overview of C-terp	266
C-terp Debugging Functions	267
The trace() Function	267
The dumpv() Function	271
The check() Function	272
The breakpt() Function	274
Debug Mode	274
(D/d)isplay	274
Traceback	275
(N/n)ext-Step	275
(S/s)ide-step	275
Continue	275
Restore, Flip, Window, and Edit	276
Using C-terp	276
Selecting Breakpoints by Function Name	278
Finding Pointers that Write on Data	280

Building a Watch Function	281
Bottom-up Testing with C-terp	282
RETURN	283

11 Conclusion

Looking Back	285
The Formal Framework	286
Universal Techniques	286
C's Unique Problems	286
Arcane Tricks	286
Sophisticated Systems	286
A Curious State of Affairs	287
A Challenge to Compiler Vendors	287
A Challenge to the Reader	288

A Resource List	289
------------------------------	------------

B A Full-Featured Debugging System	293
---	------------

C Assembly Language Functions	319
--	------------

D A Poor Man's Ctrace	323
------------------------------------	------------

E sdb Command Summary	337
------------------------------------	------------

Bibliography	341
---------------------------	------------

Index	343
--------------------	------------

Introduction

Why a Book on Debugging?

To write programs that work, you must know how to debug. It's that simple. In fact, if you produce working programs, you will spend at least half your time debugging.

To write programs that work, you must know also how to define problems (systems analysis), how to design solutions (algorithms and software engineering), and how to code in your chosen programming language (syntax and good practice). There are hundreds of books written on these subjects.

Isn't it strange that there are no books on debugging? I think so. All programmers do it, but nobody wants to talk about it and I think that's a mistake. First, you *can* improve your debugging skills. Like design skills, debugging skills evolve from critical analysis and exposure to new techniques and ideas. Second, you *must* improve your debugging skills. Good debugging skills are requisite to successful programming. To grow as a programmer and to tackle increasingly challenging assignments, your ability to debug also must grow. Because you program in C, you'll face peculiar debugging problems that demand enhanced debugging skills.

Debugging Then and Debugging Now

When I began programming in 1969, the teaching language was Fortran IV. A well-designed program was one that got the right answer. Attitudes about program design have changed since then. Today, by means of such carefully designed teaching languages

as Pascal and Logo, beginning students are introduced early to modular design. But, when I consider how I learned to debug, my introduction to programming seems rigorously structured by comparison. Except for a vague admonition to “use trace statements to find any problems,” I don’t remember any classroom advice about how to debug. Sadly, that situation hasn’t changed much.

The popular introductory Pascal programming texts (*Oh! Pascal!*, by Doug Cooper and Michael Clancy; *Introduction to Pascal and Structured Design*, by Nell Dale and David Orshalick; and *An Introduction to Programming and Problem Solving with Pascal*, 2nd Edition, by G. Michael Schneider) include debugging tips—usually at the end of each chapter. The Clancy and Cooper sections are called “debugging and anti-bugging.” I suggest that a more precise title might be “anti-bugging and anti-bugging.” Except for the all-purpose admonition to include trace statements, the book sections cover defensive programming strategies, rather than debugging strategies. The authors nurture a belief that either good programmers won’t have bugs or, if they do, programmers know intuitively how to find bugs.

Until I started teaching, I assumed that good debugging skills were a natural outgrowth of good design skills. Not so. A bright student may intuitively decompose a problem into beautifully coherent, cohesive, functional modules. That same student may not be able to find the most trivial syntax errors, let alone discover subtle run-time bugs. Equally bright students turn in working designs that literally defy analysis. While I don’t believe that we learn debugging by studying design, I do believe that we can learn efficient debugging.

We can develop a methodological model that directs our efforts toward more productive searches. We can acquire heuristic knowledge (a kind of folk wisdom) about where to look first. We can be deliberately sensitive to the different variables and observable phenomena in different environments. We can become expert at selecting and using appropriate tools. And, through critical analysis of our attempts to find “worthy” bugs, we can learn from our own mistakes.

Debugging = Working Programs

The simple truth is that programs seldom run right the first time. As I pointed out, debugging accounts for at least half of most develop-

ment project time. Careful, disciplined design and coding won't eliminate the need for debugging; good design and coding simply make efficient debugging possible.

There is no technological relief on the horizon. The design/debug effort ratio remains fairly constant across all language classes. Neither better design nor better languages will eliminate debugging. Although debugging may change shape—we may someday find ourselves debugging specifications rather than procedures—we will need debugging as long as we create new applications.

As a teacher, I've come to believe that the greatest single difference between the person who succeeds in a programming class and the person who doesn't is that the successful programmer develops debugging skills and the unsuccessful programmer does not. While all learners commit the same types of errors in their first drafts, the individual who doesn't understand debugging remains blocked and frustrated, and the individual who does understand debugging finds the errors and goes on to experiment with new and more powerful techniques.

You may discover that your growth as a programmer has been restricted by your debugging skills. Think of a time when you were frustrated by a new operating system or language. Was part of your difficulty caused by your inability to find bugs in the new environment? To restate the question, if I could guarantee that you would find a new bug after only two seconds of searching, would you be willing to tackle almost any project in any environment? Successful debugging supports exploration and refines understanding. The more adept you are at finding bugs in a given environment, the more readily you can master that environment.

Debugging C Is Demanding

You can't rely on debugging techniques borrowed from other programming languages when you program in C. C programmers willingly discard the protection provided by other high-level languages. Too often, newcomers underestimate the impact of this change on the debugging environment. Novices assume that because both C and Pascal are small and well-structured languages, debugging C is similar to debugging Pascal. Nothing could be farther from the truth. Some of the special problems associated