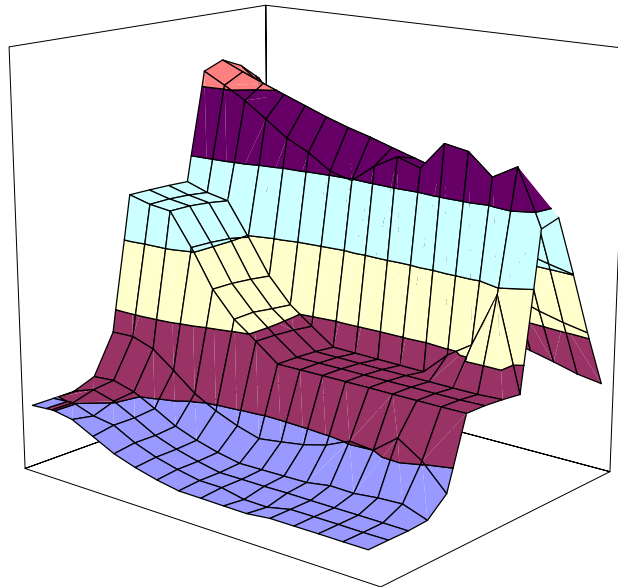

Computer Systems
*A Programmer's Perspective*¹
(Beta Draft)



Randal E. Bryant
David R. O'Hallaron

November 16, 2001

¹Copyright © 2001, R. E. Bryant, D. R. O'Hallaron. All rights reserved.

Contents

Preface	i
1 Introduction	1
1.1 Information is Bits in Context	2
1.2 Programs are Translated by Other Programs into Different Forms	3
1.3 It Pays to Understand How Compilation Systems Work	4
1.4 Processors Read and Interpret Instructions Stored in Memory	5
1.4.1 Hardware Organization of a System	5
1.4.2 Running the <code>hello</code> Program	8
1.5 Caches Matter	9
1.6 Storage Devices Form a Hierarchy	10
1.7 The Operating System Manages the Hardware	11
1.7.1 Processes	13
1.7.2 Threads	14
1.7.3 Virtual Memory	14
1.7.4 Files	15
1.8 Systems Communicate With Other Systems Using Networks	16
1.9 Summary	18
I Program Structure and Execution	19
2 Representing and Manipulating Information	21
2.1 Information Storage	22
2.1.1 Hexadecimal Notation	23
2.1.2 Words	25

2.1.3	Data Sizes	25
2.1.4	Addressing and Byte Ordering	26
2.1.5	Representing Strings	33
2.1.6	Representing Code	33
2.1.7	Boolean Algebras and Rings	34
2.1.8	Bit-Level Operations in C	37
2.1.9	Logical Operations in C	39
2.1.10	Shift Operations in C	40
2.2	Integer Representations	41
2.2.1	Integral Data Types	41
2.2.2	Unsigned and Two's Complement Encodings	41
2.2.3	Conversions Between Signed and Unsigned	45
2.2.4	Signed vs. Unsigned in C	47
2.2.5	Expanding the Bit Representation of a Number	49
2.2.6	Truncating Numbers	51
2.2.7	Advice on Signed vs. Unsigned	52
2.3	Integer Arithmetic	53
2.3.1	Unsigned Addition	53
2.3.2	Two's Complement Addition	56
2.3.3	Two's Complement Negation	60
2.3.4	Unsigned Multiplication	61
2.3.5	Two's Complement Multiplication	62
2.3.6	Multiplying by Powers of Two	63
2.3.7	Dividing by Powers of Two	64
2.4	Floating Point	66
2.4.1	Fractional Binary Numbers	67
2.4.2	IEEE Floating-Point Representation	69
2.4.3	Example Numbers	71
2.4.4	Rounding	74
2.4.5	Floating-Point Operations	76
2.4.6	Floating Point in C	77
2.5	Summary	79

3	Machine-Level Representation of C Programs	89
3.1	A Historical Perspective	90
3.2	Program Encodings	92
3.2.1	Machine-Level Code	93
3.2.2	Code Examples	94
3.2.3	A Note on Formatting	97
3.3	Data Formats	98
3.4	Accessing Information	99
3.4.1	Operand Specifiers	100
3.4.2	Data Movement Instructions	102
3.4.3	Data Movement Example	103
3.5	Arithmetic and Logical Operations	105
3.5.1	Load Effective Address	106
3.5.2	Unary and Binary Operations	106
3.5.3	Shift Operations	107
3.5.4	Discussion	108
3.5.5	Special Arithmetic Operations	109
3.6	Control	110
3.6.1	Condition Codes	110
3.6.2	Accessing the Condition Codes	111
3.6.3	Jump Instructions and their Encodings	114
3.6.4	Translating Conditional Branches	117
3.6.5	Loops	119
3.6.6	Switch Statements	128
3.7	Procedures	132
3.7.1	Stack Frame Structure	132
3.7.2	Transferring Control	134
3.7.3	Register Usage Conventions	135
3.7.4	Procedure Example	137
3.7.5	Recursive Procedures	140
3.8	Array Allocation and Access	142
3.8.1	Basic Principles	143
3.8.2	Pointer Arithmetic	144

3.8.3	Arrays and Loops	145
3.8.4	Nested Arrays	145
3.8.5	Fixed Size Arrays	148
3.8.6	Dynamically Allocated Arrays	150
3.9	Heterogeneous Data Structures	153
3.9.1	Structures	153
3.9.2	Unions	156
3.10	Alignment	160
3.11	Putting it Together: Understanding Pointers	162
3.12	Life in the Real World: Using the GDB Debugger	165
3.13	Out-of-Bounds Memory References and Buffer Overflow	167
3.14	*Floating-Point Code	172
3.14.1	Floating-Point Registers	172
3.14.2	Extended-Precision Arithmetic	173
3.14.3	Stack Evaluation of Expressions	176
3.14.4	Floating-Point Data Movement and Conversion Operations	179
3.14.5	Floating-Point Arithmetic Instructions	181
3.14.6	Using Floating Point in Procedures	183
3.14.7	Testing and Comparing Floating-Point Values	184
3.15	*Embedding Assembly Code in C Programs	186
3.15.1	Basic Inline Assembly	187
3.15.2	Extended Form of asm	189
3.16	Summary	192
4	Processor Architecture	201
5	Optimizing Program Performance	203
5.1	Capabilities and Limitations of Optimizing Compilers	204
5.2	Expressing Program Performance	207
5.3	Program Example	209
5.4	Eliminating Loop Inefficiencies	212
5.5	Reducing Procedure Calls	216
5.6	Eliminating Unneeded Memory References	218

5.7	Understanding Modern Processors	220
5.7.1	Overall Operation	221
5.7.2	Functional Unit Performance	224
5.7.3	A Closer Look at Processor Operation	225
5.8	Reducing Loop Overhead	233
5.9	Converting to Pointer Code	238
5.10	Enhancing Parallelism	241
5.10.1	Loop Splitting	241
5.10.2	Register Spilling	245
5.10.3	Limits to Parallelism	247
5.11	Putting it Together: Summary of Results for Optimizing Combining Code	247
5.11.1	Floating-Point Performance Anomaly	248
5.11.2	Changing Platforms	249
5.12	Branch Prediction and Misprediction Penalties	249
5.13	Understanding Memory Performance	252
5.13.1	Load Latency	253
5.13.2	Store Latency	255
5.14	Life in the Real World: Performance Improvement Techniques	260
5.15	Identifying and Eliminating Performance Bottlenecks	261
5.15.1	Program Profiling	261
5.15.2	Using a Profiler to Guide Optimization	263
5.15.3	Amdahl's Law	266
5.16	Summary	267
6	The Memory Hierarchy	275
6.1	Storage Technologies	276
6.1.1	Random-Access Memory	276
6.1.2	Disk Storage	285
6.1.3	Storage Technology Trends	293
6.2	Locality	295
6.2.1	Locality of References to Program Data	295
6.2.2	Locality of Instruction Fetches	297
6.2.3	Summary of Locality	297

6.3	The Memory Hierarchy	298
6.3.1	Caching in the Memory Hierarchy	301
6.3.2	Summary of Memory Hierarchy Concepts	303
6.4	Cache Memories	304
6.4.1	Generic Cache Memory Organization	305
6.4.2	Direct-Mapped Caches	306
6.4.3	Set Associative Caches	313
6.4.4	Fully Associative Caches	315
6.4.5	Issues with Writes	318
6.4.6	Instruction Caches and Unified Caches	319
6.4.7	Performance Impact of Cache Parameters	320
6.5	Writing Cache-friendly Code	322
6.6	Putting it Together: The Impact of Caches on Program Performance	327
6.6.1	The Memory Mountain	327
6.6.2	Rearranging Loops to Increase Spatial Locality	331
6.6.3	Using Blocking to Increase Temporal Locality	335
6.7	Summary	338
II	Running Programs on a System	347
7	Linking	349
7.1	Compiler Drivers	350
7.2	Static Linking	351
7.3	Object Files	352
7.4	Relocatable Object Files	353
7.5	Symbols and Symbol Tables	354
7.6	Symbol Resolution	357
7.6.1	How Linkers Resolve Multiply-Defined Global Symbols	358
7.6.2	Linking with Static Libraries	361
7.6.3	How Linkers Use Static Libraries to Resolve References	364
7.7	Relocation	365
7.7.1	Relocation Entries	366
7.7.2	Relocating Symbol References	367

7.8	Executable Object Files	371
7.9	Loading Executable Object Files	372
7.10	Dynamic Linking with Shared Libraries	374
7.11	Loading and Linking Shared Libraries from Applications	376
7.12	*Position-Independent Code (PIC)	377
7.13	Tools for Manipulating Object Files	381
7.14	Summary	382
8	Exceptional Control Flow	391
8.1	Exceptions	392
8.1.1	Exception Handling	393
8.1.2	Classes of Exceptions	394
8.1.3	Exceptions in Intel Processors	397
8.2	Processes	398
8.2.1	Logical Control Flow	398
8.2.2	Private Address Space	399
8.2.3	User and Kernel Modes	400
8.2.4	Context Switches	401
8.3	System Calls and Error Handling	402
8.4	Process Control	403
8.4.1	Obtaining Process ID's	404
8.4.2	Creating and Terminating Processes	404
8.4.3	Reaping Child Processes	409
8.4.4	Putting Processes to Sleep	414
8.4.5	Loading and Running Programs	415
8.4.6	Using <code>fork</code> and <code>execve</code> to Run Programs	418
8.5	Signals	419
8.5.1	Signal Terminology	423
8.5.2	Sending Signals	423
8.5.3	Receiving Signals	426
8.5.4	Signal Handling Issues	429
8.5.5	Portable Signal Handling	434
8.6	Nonlocal Jumps	436

8.7	Tools for Manipulating Processes	441
8.8	Summary	441
9	Measuring Program Execution Time	449
9.1	The Flow of Time on a Computer System	450
9.1.1	Process Scheduling and Timer Interrupts	451
9.1.2	Time from an Application Program's Perspective	452
9.2	Measuring Time by Interval Counting	454
9.2.1	Operation	456
9.2.2	Reading the Process Timers	456
9.2.3	Accuracy of Process Timers	457
9.3	Cycle Counters	459
9.3.1	IA32 Cycle Counters	460
9.4	Measuring Program Execution Time with Cycle Counters	460
9.4.1	The Effects of Context Switching	462
9.4.2	Caching and Other Effects	463
9.4.3	The <i>K</i> -Best Measurement Scheme	467
9.5	Time-of-Day Measurements	476
9.6	Putting it Together: An Experimental Protocol	478
9.7	Looking into the Future	480
9.8	Life in the Real World: An Implementation of the <i>K</i> -Best Measurement Scheme	480
9.9	Summary	481
10	Virtual Memory	485
10.1	Physical and Virtual Addressing	486
10.2	Address Spaces	487
10.3	VM as a Tool for Caching	488
10.3.1	DRAM Cache Organization	489
10.3.2	Page Tables	489
10.3.3	Page Hits	490
10.3.4	Page Faults	491
10.3.5	Allocating Pages	492
10.3.6	Locality to the Rescue Again	493

10.4 VM as a Tool for Memory Management	493
10.4.1 Simplifying Linking	494
10.4.2 Simplifying Sharing	494
10.4.3 Simplifying Memory Allocation	495
10.4.4 Simplifying Loading	495
10.5 VM as a Tool for Memory Protection	496
10.6 Address Translation	497
10.6.1 Integrating Caches and VM	500
10.6.2 Speeding up Address Translation with a TLB	500
10.6.3 Multi-level Page Tables	501
10.6.4 Putting it Together: End-to-end Address Translation	504
10.7 Case Study: The Pentium/Linux Memory System	508
10.7.1 Pentium Address Translation	508
10.7.2 Linux Virtual Memory System	513
10.8 Memory Mapping	516
10.8.1 Shared Objects Revisited	517
10.8.2 The <code>fork</code> Function Revisited	519
10.8.3 The <code>execve</code> Function Revisited	519
10.8.4 User-level Memory Mapping with the <code>mmap</code> Function	520
10.9 Dynamic Memory Allocation	522
10.9.1 The <code>malloc</code> and <code>free</code> Functions	523
10.9.2 Why Dynamic Memory Allocation?	524
10.9.3 Allocator Requirements and Goals	526
10.9.4 Fragmentation	528
10.9.5 Implementation Issues	529
10.9.6 Implicit Free Lists	529
10.9.7 Placing Allocated Blocks	531
10.9.8 Splitting Free Blocks	531
10.9.9 Getting Additional Heap Memory	532
10.9.10 Coalescing Free Blocks	532
10.9.11 Coalescing with Boundary Tags	533
10.9.12 Putting it Together: Implementing a Simple Allocator	535
10.9.13 Explicit Free Lists	543

10.9.14 Segregated Free Lists	544
10.10 Garbage Collection	546
10.10.1 Garbage Collector Basics	547
10.10.2 Mark&Sweep Garbage Collectors	548
10.10.3 Conservative Mark&Sweep for C Programs	550
10.11 Common Memory-related Bugs in C Programs	551
10.11.1 Dereferencing Bad Pointers	551
10.11.2 Reading Uninitialized Memory	551
10.11.3 Allowing Stack Buffer Overflows	552
10.11.4 Assuming that Pointers and the Objects they Point to Are the Same Size	552
10.11.5 Making Off-by-one Errors	553
10.11.6 Referencing a Pointer Instead of the Object it Points to	553
10.11.7 Misunderstanding Pointer Arithmetic	554
10.11.8 Referencing Non-existent Variables	554
10.11.9 Referencing Data in Free Heap Blocks	555
10.11.10 Introducing Memory Leaks	555
10.12 Summary	556
III Interaction and Communication Between Programs	561
11 Concurrent Programming with Threads	563
11.1 Basic Thread Concepts	563
11.2 Thread Control	566
11.2.1 Creating Threads	567
11.2.2 Terminating Threads	567
11.2.3 Reaping Terminated Threads	568
11.2.4 Detaching Threads	568
11.3 Shared Variables in Threaded Programs	570
11.3.1 Threads Memory Model	570
11.3.2 Mapping Variables to Memory	570
11.3.3 Shared Variables	572
11.4 Synchronizing Threads with Semaphores	573
11.4.1 Sequential Consistency	573

11.4.2	Progress Graphs	576
11.4.3	Protecting Shared Variables with Semaphores	579
11.4.4	Posix Semaphores	580
11.4.5	Signaling With Semaphores	581
11.5	Synchronizing Threads with Mutex and Condition Variables	583
11.5.1	Mutex Variables	583
11.5.2	Condition Variables	586
11.5.3	Barrier Synchronization	587
11.5.4	Timeout Waiting	588
11.6	Thread-safe and Reentrant Functions	592
11.6.1	Reentrant Functions	593
11.6.2	Thread-safe Library Functions	596
11.7	Other Synchronization Errors	596
11.7.1	Races	596
11.7.2	Deadlocks	599
11.8	Summary	600
12	Network Programming	605
12.1	Client-Server Programming Model	605
12.2	Networks	606
12.3	The Global IP Internet	611
12.3.1	IP Addresses	612
12.3.2	Internet Domain Names	614
12.3.3	Internet Connections	618
12.4	Unix file I/O	619
12.4.1	The <code>read</code> and <code>write</code> Functions	620
12.4.2	Robust File I/O With the <code>readn</code> and <code>writen</code> Functions.	621
12.4.3	Robust Input of Text Lines Using the <code>readline</code> Function	623
12.4.4	The <code>stat</code> Function	623
12.4.5	The <code>dup2</code> Function	626
12.4.6	The <code>close</code> Function	627
12.4.7	Other Unix I/O Functions	628
12.4.8	Unix I/O vs. Standard I/O	628

12.5	The Sockets Interface	629
12.5.1	Socket Address Structures	629
12.5.2	The <code>socket</code> Function	631
12.5.3	The <code>connect</code> Function	631
12.5.4	The <code>bind</code> Function	633
12.5.5	The <code>listen</code> Function	633
12.5.6	The <code>accept</code> Function	635
12.5.7	Example Echo Client and Server	636
12.6	Concurrent Servers	638
12.6.1	Concurrent Servers Based on Processes	638
12.6.2	Concurrent Servers Based on Threads	640
12.7	Web Servers	646
12.7.1	Web Basics	647
12.7.2	Web Content	647
12.7.3	HTTP Transactions	648
12.7.4	Serving Dynamic Content	651
12.8	Putting it Together: The TINY Web Server	652
12.9	Summary	662
A	Error handling	665
A.1	Introduction	665
A.2	Error handling in Unix systems	666
A.3	Error-handling wrappers	667
A.4	The <code>csapp.h</code> header file	671
A.5	The <code>csapp.c</code> source file	675
B	Solutions to Practice Problems	691
B.1	Intro	691
B.2	Representing and Manipulating Information	691
B.3	Machine Level Representation of C Programs	700
B.4	Processor Architecture	715
B.5	Optimizing Program Performance	715
B.6	The Memory Hierarchy	717

CONTENTS

15

B.7 Linking 723

B.8 Exceptional Control Flow 725

B.9 Measuring Program Performance 728

B.10 Virtual Memory 730

B.11 Concurrent Programming with Threads 734

B.12 Network Programming 736

Preface

This book is for programmers who want to improve their skills by learning about what is going on “under the hood” of a computer system. Our aim is to explain the important and enduring concepts underlying all computer systems, and to show you the concrete ways that these ideas affect the correctness, performance, and utility of your application programs. By studying this book, you will gain some insights that have immediate value to you as a programmer, and others that will prepare you for advanced courses in compilers, computer architecture, operating systems, and networking.

The book owes its origins to an introductory course that we developed at Carnegie Mellon in the Fall of 1998, called *15-213: Introduction to Computer Systems*. The course has been taught every semester since then, each time to about 150 students, mostly sophomores in computer science and computer engineering. It has become a prerequisite for all upper-level systems courses. The approach is concrete and hands-on. Because of this, we are able to couple the lectures with programming labs and assignments that are fun and exciting.

The response from our students and faculty colleagues was so overwhelming that we decided that others might benefit from our approach. Hence the book. This is the Beta draft of the manuscript. The final hard-cover version will be available from the publisher in Summer, 2002, for adoption in the Fall, 2002 term.

Assumptions About the Reader’s Background

This course is based on Intel-compatible processors (called “IA32” by Intel and “x86” colloquially) running C programs on the Unix operating system. The text contains numerous programming examples that have been compiled and run under Unix. We assume that you have access to such a machine, and are able to log in and do simple things such as changing directories. Even if you don’t use Linux, much of the material applies to other systems as well. Intel-compatible processors running one of the Windows operating systems use the same instruction set, and support many of the same programming libraries. By getting a copy of the Cygwin tools (<http://cygwin.com/>), you can set up a Unix-like shell under Windows and have an environment very close to that provided by Unix.

We also assume that you have some familiarity with C or C++. If your only prior experience is with Java, the transition will require more effort on your part, but we will help you. Java and C share similar syntax and control statements. However, there are aspects of C, particularly pointers, explicit dynamic memory allocation, and formatted I/O, that do not exist in Java. The good news is that C is a small language, and it

is clearly and beautifully described in the classic “K&R” text by Brian Kernighan and Dennis Ritchie [37]. Regardless of your programming background, consider K&R an essential part of your personal library.

New to C?

To help readers whose background in C programming is weak (or nonexistent), we have included these special notes to highlight features that are especially important in C. We assume you are familiar with C++ or Java. **End**

Several of the early chapters in our book explore the interactions between C programs and their machine-language counterparts. The machine language examples were all generated by the GNU GCC compiler running on an Intel IA32 processor. We do not assume any prior experience with hardware, machine language, or assembly-language programming.

How to Read This Book

Learning how computer systems work from a programmer’s perspective is great fun, mainly because it can be done so actively. Whenever you learn some new thing, you can try it out right away and see the result first hand. In fact, we believe that the only way to learn systems is to *do* systems, either working concrete problems, or writing and running programs on real systems.

This theme pervades the entire book. When a new concept is introduced, it is followed in the text by one or more *Practice Problems* that you should work immediately to test your understanding. Solutions to the Practice Problems are at the back of the book. As you read, try to solve each problem on your own, and then check the solution to make sure you’re on the right track. Each chapter is followed by a set of *Homework Problems* of varying difficulty. Your instructor has the solutions to the Homework Problems in an Instructor’s Manual. Each Homework Problem is classified according to how much work it will be:

Category 1: Simple, quick problem to try out some idea in the book.

Category 2: Requires 5–15 minutes to complete, perhaps involving writing or running programs.

Category 3: A sustained problem that might require hours to complete.

Category 4: A laboratory assignment that might take one or two weeks to complete.

Each code example in the text was formatted directly, without any manual intervention, from a C program compiled with GCC version 2.95.3, and tested on a Linux system with a 2.2.16 kernel. The programs are available from our Web page at www.cs.cmu.edu/~ics.

The file names of the larger programs are documented in horizontal bars that surround the formatted code. For example, the program

code/intro/hello.c

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("hello, world\n");
6 }
```

code/intro/hello.c

can be found in the file `hello.c` in directory `code/intro/`. We strongly encourage you to try running the example programs on your system as you encounter them.

There are various places in the book where we show you how to run programs on Unix systems:

```
unix> ./hello
hello, world
unix>
```

In all of our examples, the output is displayed in a roman font, and the input that you type is displayed in an italicized font. In this particular example, the Unix shell program prints a command-line prompt and waits for you to type something. After you type the string `./hello` and hit the return or enter key, the shell loads and runs the `hello` program from the current directory. The program prints the string `hello, world\n` and terminates. Afterwards, the shell prints another prompt and waits for the next command. The vast majority of our examples do not depend on any particular version of Unix, and we indicate this independence with the generic `unix>` prompt. In the rare cases where we need to make a point about a particular version of Unix such as Linux or Solaris, we include its name in the command-line prompt.

Finally, some sections (denoted by a `“*”`) contain material that you might find interesting, but that can be skipped without any loss of continuity.

Acknowledgements

We are deeply indebted to many friends and colleagues for their thoughtful criticisms and encouragement. A special thanks to our 15-213 students, whose infectious energy and enthusiasm spurred us on. Nick Carter and Vinny Furia generously provided their malloc package. Chris Lee, Mathilde Pignol, and Zia Khan identified typos in early drafts.

Guy Blelloch, Bruce Maggs, and Todd Mowry taught the course over multiple semesters, gave us encouragement, and helped improve the course material. Herb Derby provided early spiritual guidance and encouragement. Allan Fisher, Garth Gibson, Thomas Gross, Satya, Peter Steenkiste, and Hui Zhang encouraged us to develop the course from the start. A suggestion from Garth early on got the whole ball rolling, and this was picked up and refined with the help of a group led by Allan Fisher. Mark Stehlik and Peter Lee have been very supportive about building this material into the undergraduate curriculum. Greg Kesden provided

helpful feedback. Greg Ganger and Jiri Schindler graciously provided some disk drive characterizations and answered our questions on modern disks. Tom Stricker showed us the memory mountain.

A special group of students, Khalil Amiri, Angela Demke Brown, Chris Colohan, Jason Crawford, Peter Dinda, Julio Lopez, Bruce Lowekamp, Jeff Pierce, Sanjay Rao, Blake Scholl, Greg Steffan, Tiankai Tu, and Kip Walker, were instrumental in helping us develop the content of the course.

In particular, Chris Colohan established a fun (and funny) tone that persists to this day, and invented the legendary “binary bomb” that has proven to be a great tool for teaching machine code and debugging concepts.

Chris Bauer, Alan Cox, David Daugherty, Peter Dinda, Sandhya Dwarkadis, John Greiner, Bruce Jacob, Barry Johnson, Don Heller, Bruce Lowekamp, Greg Morrisett, Brian Noble, Bobbie Othmer, Bill Pugh, Michael Scott, Mark Smotherman, Greg Steffan, and Bob Wier took time that they didn’t have to read and advise us on early drafts of the book. A very special thanks to Peter Dinda (Northwestern University), John Greiner (Rice University), Bruce Lowekamp (William & Mary), Bobbie Othmer (University of Minnesota), Michael Scott (University of Rochester), and Bob Wier (Rocky Mountain College) for class testing the Beta version. A special thanks to their students as well!

Finally, we would like to thank our colleagues at Prentice Hall. Eric Frank (Editor) and Harold Stone (Consulting Editor) have been unflagging in their support and vision. Jerry Ralya (Development Editor) has provided sharp insights.

Thank you all.

Randy Bryant
Dave O’Hallaron

Pittsburgh, PA
Aug 1, 2001

Chapter 1

Introduction

A *computer system* is a collection of hardware and software components that work together to run computer programs. Specific implementations of systems change over time, but the underlying concepts do not. All systems have similar hardware and software components that perform similar functions. This book is written for programmers who want to improve at their craft by understanding how these components work and how they affect the correctness and performance of their programs.

In their classic text on the C programming language [37], Kernighan and Ritchie introduce readers to C using the `hello` program shown in Figure 1.1.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("hello, world\n");
6 }
```

code/intro/hello.c

code/intro/hello.c

Figure 1.1: **The `hello` program.**

Although `hello` is a very simple program, every major part of the system must work in concert in order for it to run to completion. In a sense, the goal of this book is to help you understand what happens and why, when you run `hello` on your system.

We will begin our study of systems by tracing the lifetime of the `hello` program, from the time it is created by a programmer, until it runs on a system, prints its simple message, and terminates. As we follow the lifetime of the program, we will briefly introduce the key concepts, terminology, and components that come into play. Later chapters will expand on these ideas.

1.1 Information is Bits in Context

Our `hello` program begins life as a *source program* (or *source file*) that the programmer creates with an editor and saves in a text file called `hello.c`. The source program is a sequence of bits, each with a value of 0 or 1, organized in 8-bit chunks called *bytes*. Each byte represents some text character in the program.

Most modern systems represent text characters using the ASCII standard that represents each character with a unique byte-sized integer value. For example, Figure 1.2 shows the ASCII representation of the `hello.c` program.

#	i	n	c	l	u	d	e	<sp>	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	<sp>	m	a	i	n	()	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	<sp>	<sp>	<sp>	<sp>	p	r	i	n	t	f	("	h	e	l
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108
l	o	,	<sp>	w	o	r	l	d	\	n	")	;	\n	}
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	125

Figure 1.2: The ASCII text representation of `hello.c`.

The `hello.c` program is stored in a file as a sequence of bytes. Each byte has an integer value that corresponds to some character. For example, the first byte has the integer value 35, which corresponds to the character '#'. The second byte has the integer value 105, which corresponds to the character 'i', and so on. Notice that each text line is terminated by the invisible *newline* character '\n', which is represented by the integer value 10. Files such as `hello.c` that consist exclusively of ASCII characters are known as *text files*. All other files are known as *binary files*.

The representation of `hello.c` illustrates a fundamental idea: All information in a system — including disk files, programs stored in memory, user data stored in memory, and data transferred across a network — is represented as a bunch of bits. The only thing that distinguishes different data objects is the context in which we view them. For example, in different contexts, the same sequence of bytes might represent an integer, floating point number, character string, or machine instruction. This idea is explored in detail in Chapter 2.

Aside: The C programming language.

C was developed in 1969 to 1973 by Dennis Ritchie of Bell Laboratories. The American National Standards Institute (ANSI) ratified the ANSI C standard in 1989. The standard defines the C language and a set of library functions known as the *C standard library*. Kernighan and Ritchie describe ANSI C in their classic book, which is known affectionately as “K&R” [37].

In Ritchie’s words [60], C is “quirky, flawed, and an enormous success.” So why the success?

- *C was closely tied with the Unix operating system.* C was developed from the beginning as the system programming language for Unix. Most of the Unix kernel, and all of its supporting tools and libraries, were written in C. As Unix became popular in universities in the late 1970s and early 1980s, many people were

exposed to C and found that they liked it. Since Unix was written almost entirely in C, it could be easily ported to new machines, which created an even wider audience for both C and Unix.

- *C is a small, simple language.* The design was controlled by a single person, rather than a committee, and the result was a clean, consistent design with little baggage. The K&R book describes the complete language and standard library, with numerous examples and exercises, in only 261 pages. The simplicity of C made it relatively easy to learn and to port to different computers.
- *C was designed for a practical purpose.* C was designed to implement the Unix operating system. Later, other people found that they could write the programs they wanted, without the language getting in the way.

C is the language of choice for system-level programming, and there is a huge installed based of application-level programs as well. However, it is not perfect for all programmers and all situations. C pointers are a common source of confusion and programming errors. C also lacks explicit support for useful abstractions such as classes and objects. Newer languages such as C++ and Java address these issues for application-level programs. **End Aside.**

1.2 Programs are Translated by Other Programs into Different Forms

The `hello` program begins life as a high-level C program because it can be read and understood by human beings in that form. However, in order to run `hello.c` on the system, the individual C statements must be translated by other programs into a sequence of low-level *machine-language* instructions. These instructions are then packaged in a form called an *executable object program*, and stored as a binary disk file. Object programs are also referred to as *executable object files*.

On a Unix system, the translation from source file to object file is performed by a *compiler driver*:

```
unix> gcc -o hello hello.c
```

Here, the GCC compiler driver reads the source file `hello.c` and translates it into an executable object file `hello`. The translation is performed in the sequence of four phases shown in Figure 1.3. The programs that perform the four phases (*preprocessor*, *compiler*, *assembler*, and *linker*) are known collectively as the *compilation system*.

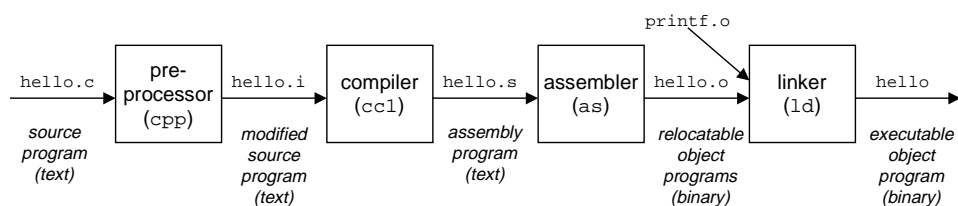


Figure 1.3: The compilation system.

- *Preprocessing phase.* The preprocessor (`cpp`) modifies the original C program according to directives that begin with the `#` character. For example, the `#include <stdio.h>` command in line 1 of `hello.c` tells the preprocessor to read the contents of the system header file `stdio.h` and insert it directly into the program text. The result is another C program, typically with the `.i` suffix.

- *Compilation phase.* The compiler (`cc1`) translates the text file `hello.i` into the text file `hello.s`, which contains an *assembly-language program*. Each statement in an assembly-language program exactly describes one low-level machine-language instruction in a standard text form. Assembly language is useful because it provides a common output language for different compilers for different high-level languages. For example, C compilers and Fortran compilers both generate output files in the same assembly language.
- *Assembly phase.* Next, the assembler (`as`) translates `hello.s` into machine-language instructions, packages them in a form known as a *relocatable object program*, and stores the result in the object file `hello.o`. The `hello.o` file is a binary file whose bytes encode machine language instructions rather than characters. If we were to view `hello.o` with a text editor, it would appear to be gibberish.
- *Linking phase.* Notice that our `hello` program calls the `printf` function, which is part of the *standard C library* provided by every C compiler. The `printf` function resides in a separate precompiled object file called `printf.o`, which must somehow be merged with our `hello.o` program. The linker (`ld`) handles this merging. The result is the `hello` file, which is an *executable object file* (or simply *executable*) that is ready to be loaded into memory and executed by the system.

Aside: The GNU project.

GCC is one of many useful tools developed by the GNU (GNU's Not Unix) project. The GNU project is a tax-exempt charity started by Richard Stallman in 1984, with the ambitious goal of developing a complete Unix-like system whose source code is unencumbered by restrictions on how it can be modified or distributed. As of 2002, the GNU project has developed an environment with all the major components of a Unix operating system, except for the kernel, which was developed separately by the Linux project. The GNU environment includes the EMACS editor, GCC compiler, GDB debugger, assembler, linker, utilities for manipulating binaries, and many others.

The GNU project is a remarkable achievement, and yet it is often overlooked. The modern open source movement (commonly associated with Linux) owes its intellectual origins to the GNU project's notion of *free software*. Further, Linux owes much of its popularity to the GNU tools, which provide the environment for the Linux kernel. **End Aside.**

1.3 It Pays to Understand How Compilation Systems Work

For simple programs such as `hello.c`, we can rely on the compilation system to produce correct and efficient machine code. However, there are some important reasons why programmers need to understand how compilation systems work:

- *Optimizing program performance.* Modern compilers are sophisticated tools that usually produce good code. As programmers, we do not need to know the inner workings of the compiler in order to write efficient code. However, in order to make good coding decisions in our C programs, we do need a basic understanding of assembly language and how the compiler translates different C statements into assembly language. For example, is a `switch` statement always more efficient than a sequence of `if-then-else` statements? Just how expensive is a function call? Is a `while` loop more efficient than a `do` loop? Are pointer references more efficient than array indexes? Why does our loop run so much faster if we sum into a local variable instead of an argument that is passed by reference? Why do two functionally equivalent loops have such different running times?

- [Zorba the Hutt's Revenge \(Star Wars\) pdf, azw \(kindle\), epub, doc, mobi](#)
- [download online The Enigma of Capital: And the Crises of Capitalism pdf](#)
- [click The O'Sullivan Twins \(St Clare's, Book 2\) pdf, azw \(kindle\), epub](#)
- [Of Blood and Honey \(A Book of the Fey and the Fallen\) here](#)
- [Kids of Kabul: Living Bravely Through a Never-Ending War pdf, azw \(kindle\)](#)

- <http://berttrotman.com/library/Conversations-with-William-Maxwell.pdf>
- <http://damianfoster.com/books/A-Companion-to-Foucault--Blackwell-Companions-to-Philosophy-.pdf>
- <http://betsy.wesleychapelcomputerrepair.com/library/The-O-Sullivan-Twins--St-Clare-s--Book-2-.pdf>
- <http://dadhoc.com/lib/Applied-Issues-in-Investigative-Interviewing--Eyewitness-Memory--and-Credibility-Assessment.pdf>
- <http://transtrade.cz/?ebooks/Managing-Depression-with-Mindfulness-for-Dummies.pdf>