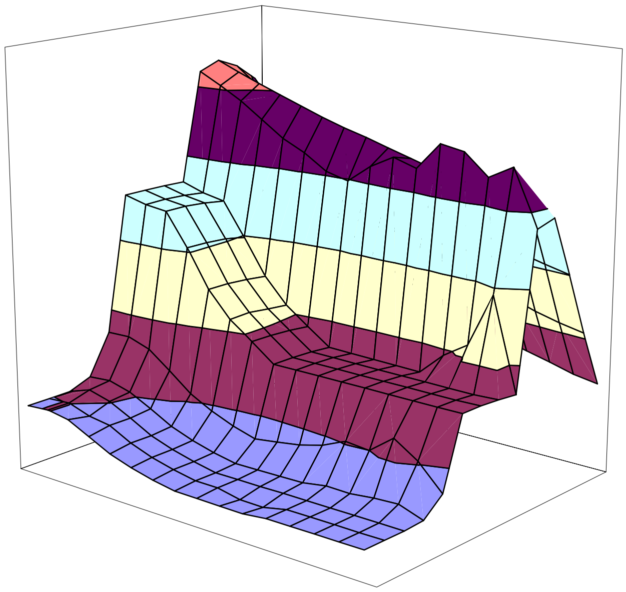
Computer Systems

A Programmer's Perspective [1]

(*Beta Draft*)



Randal E. Bryant
David R. O'Hallaron

November 16, 2001

Contents

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*

```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
}
```

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.

code/intro/hello.c

```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
}
```

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 understand 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*.

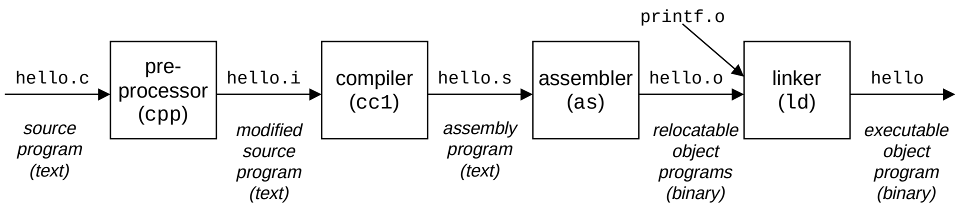


Figure 1.3: **The compilation system.**

- *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?