Preface

This book (CS:APP) is for programmers who want to improve their skills by learning what is going on "under the hood" of a computer system.

Our aim is to explain the 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. Unlike other systems books, which are written primarily for system builders, this book is written for programmers, from a programmer's perspective.

If you study and learn the concepts in this book, you will be on your way to becoming the rare "power programmer" who knows how things work and how to fix them when they break. You will also be prepared to study specific systems topics such as compilers, computer architecture, operating systems, embedded systems, and networking.

Assumptions About the Reader's Background

The examples in the book are based on Intel-compatible processors (called "IA32" by Intel and "x86" colloquially) running C programs on Unix or Unix-like (such as Linux) operating systems. (To simplify our presentation, we will use the term "Unix" as an umbrella term for systems like Solaris and Linux.) The text contains numerous programming examples that have been compiled and run on Linux systems. We assume that you have access to such a machine, and are able to log in and do simple things such as changing directories.

If your computer runs Microsoft Windows, you have two choices. First, you can get a copy of Linux (see `www.linux.org` or `www.redhat.com`) and install it as a "dual boot" option, so that your machine can run either operating system. Alternatively, by installing a copy of the Cygwin tools (`www.cygwin.com`), you can have up a Unix-like shell under Windows and have an environment very close to that provided by Linux. Not all features of Linux are available under Cygwin, however.

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. Fortunately, C is a small language, and it is clearly and beautifully described in the classic "K&R" text by Brian Kernighan and Dennis Ritchie [40]. Regardless of your programming background, consider K&R an essential part of your personal systems library.

Several of the early chapters in the 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.

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

How to Read the 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 end of each chapter (look for the blue edge). As you read, try to solve each problem on your own, and then check the solution to make sure you are 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. For each homework problem, we show a rating of the amount of effort we feel it will require:

♦	Should require just a few minutes. Little or no programming required.
♦♦	Might require up to 20 minutes. Often involves writing and testing some code. Many of these are derived from problems we have given on exams.
♦♦♦	Requires a significant effort, perhaps 1–2 hours. Generally involves writing and testing a significant amount of code.
♦♦♦♦	A lab assignment, requiring up to 10 hours of effort.

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. All of the source code is available from the CS:APP Web page at `csapp.cs.cmu.edu`. In the text, the file names of the source programs are documented in horizontal bars that surround the formatted code. For example, the program in Figure 1 can be found in the file `hello.c` in directory `code/intro/`. We encourage you to try running the example programs on your system as you encounter them.

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

Aside: What is an aside?
You will encounter asides of this form throughout the text. Asides are parenthetical remarks that give you some additional insight into the current topic. Asides serve a number of purposes. Some are little history lessons. For

code/intro/hello.c

```
#include <stdio.h>

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

code/intro/hello.c

Figure 1: **A typical code example.**

example, where did C, Linux, and the Internet come from? Other asides are meant to clarify ideas that students often find confusing. For example, what is the difference between a cache line, set, and block? Other asides give real-world examples. For example, how a floating-point error crashed a French rocket, or what the geometry of a real IBM disk drive looks like. Finally, some asides are just fun stuff. For example, what is a "hoinky"? **End Aside.**

Origins of the Book

The book stems from an introductory course that we developed at Carnegie Mellon University in the Fall of 1998, called *15-213: Introduction to Computer Systems* (ICS) [7]. The ICS course has been taught every semester since then, each time to about 150 students, mostly sophomores in computer science and computer engineering. It has since become a prerequisite for most upper-level systems courses in the CS and ECE departments at Carnegie Mellon.

The idea with ICS was to introduce students to computers in a different way. Few of our students would have the opportunity to build a computer system. On the other hand, most students, even the computer engineers, would be required to use and program computers on a daily basis. So we decided to teach about systems from the point of view of the programmer, using the following filter: We would cover a topic only if it affected the performance, correctness, or utility of user-level C programs.

For example, topics such as hardware adder and bus designs were out. Topics such as machine language were in, but instead of focusing on how to write assembly language, we would look at how C constructs such as pointers, loops, procedure calls and returns, and switch statements were translated by the compiler. Further, we would take a broader and more realistic view of the system as both hardware and systems software, covering such topics as linking, loading, processes, signals, performance optimization, measurement, I/O, and network and concurrent programming.

This approach allowed us to teach the ICS course in a way that was practical, concrete, hands-on, and exciting for the students. The response from our students and faculty colleagues was immediate and overwhelmingly positive, and we realized that others outside of CMU might benefit from using our approach. Hence this book, which we developed over a period of two years from the ICS lecture notes.

Aside: ICS numerology.
The numerology of the ICS course is a little strange. About halfway through the first semester, we realized that the assigned course number (15-213) was also the CMU zip code, hence the motto "15-213: The course that gives CMU its zip!". By chance, the alpha version of the manuscript was printed on February 13, 2001 (2/13/01). When

we presented the course at the SIGCSE education conference, the talk was scheduled in Room 213. And the final version of the book has 13 chapters. It's a good thing we're not superstitious! **End Aside.**

Overview of the Book

The CS:APP book consists of 13 chapters designed to capture the core ideas in computer systems:

- *Chapter 1: A Tour of Computer Systems.* This chapter introduces the major ideas and themes in computer systems by tracing the life cycle of a simple "hello, world" program.
- *Chapter 2: Representing and Manipulating Information.* We cover computer arithmetic, emphasizing the properties of unsigned and two's complement number representations that affect programmers. We consider how numbers are represented and therefore what range of values can be encoded for a given word size. We consider the effect of casting between signed and unsigned numbers. We cover the mathematical properties of arithmetic operations. Students are surprised to learn that the (two's complement) sum or product of two positive numbers can be negative. On the other hand, two's complement arithmetic satisfies ring properties, and hence a compiler can transform multiplication by a constant into a sequence of shifts and adds. We use the bit-level operations of C to demonstrate the principles and applications of Boolean algebra. We cover the IEEE floating point format in terms of how it represents values and the mathematical properties of floating point operations.

  Having a solid understanding of computer arithmetic is critical to writing reliable programs. For example, one cannot replace the expression `(x<y)` with `(x-y<0)` due to the possibility of overflow. One cannot even replace it with the expression `(-y<-x)` due to the asymmetric range of negative and positive numbers in the two's complement representation. Arithmetic overflow is a common source of programming errors, yet few other books cover the properties of computer arithmetic from a programmer's perspective.
- *Chapter 3: Machine-Level Representation of Programs.* We teach students how to read the IA32 assembly language generated by a C compiler. We cover the basic instruction patterns generated for different control constructs, such as conditionals, loops, and switch statements. We cover the implementation of procedures, including stack allocation, register usage conventions and parameter passing. We cover the way different data structures such as structures, unions, and arrays are allocated and accessed. Learning the concepts in this chapter helps students become better programmers, because they understand how their programs are represented on the machine. Another nice benefit is that students develop a concrete understanding of pointers.
- *Chapter 4: Processor Architecture.* This chapter covers basic combinational and sequential logic elements and then shows how these elements can be combined in a datapath that executes a simplified subset of the IA32 instruction set called "Y86." We begin with the design of a single-cycle non-pipelined datapath, which we extend into a five-stage pipelined design. The control logic for the processor designs in this chapter are described using a simple hardware description language called HCL. Hardware designs written in HCL can be compiled and linked into graphical processor simulators provided with the textbook.

- *Chapter 5: Optimizing Program Performance.* In this chapter we introduce a number of techniques for improving code performance. We start with machine-independent program transformations that should be standard practice when writing any program on any machine. We then progress to transformations whose efficacy depends on the characteristics of the target machine and compiler. To motivate these transformation, we introduce a simple operational model of how modern out-of-order processors work, and then show students how to use this model to improve the performance of their C programs.

- *Chapter 6: The Memory Hierarchy.* The memory system is one of the most visible parts of a computer system to application programmers. To this point, the students have relied on a conceptual model of the memory system as a linear array with uniform access times. In practice, a memory system is a hierarchy of storage devices with different capacities, costs, and access times. We cover the different types of RAM and ROM memories and the geometry and organization of modern disk drives. We describe how these storage devices are arranged in a hierarchy. We show how this hierarchy is made possible by locality of reference. We make these ideas concrete by introducing a unique view of a memory system as a "memory mountain" with ridges of temporal locality and slopes of spatial locality. Finally, we show students how to improve the performance of application programs by improving their temporal and spatial locality.

- *Chapter 7: Linking.* This chapter covers both static and dynamic linking, including the ideas of relocatable and executable object files, symbol resolution, relocation, static libraries, shared object libraries, and position-independent code. Linking is not covered in most systems texts, but we cover it for several reasons. First, some of the most confusing errors that students can encounter are related to glitches during linking, especially for large software packages. Second, the object files produced by linkers are tied to concepts such as loading, virtual memory, and memory mapping.

- *Chapter 8: Exceptional Control Flow.* In this part of the course we break the single-program model by introducing the general concept of exceptional control flow (i.e., changes in control flow that are outside the normal branches and procedure calls). We cover examples of exceptional control flow that exist at all levels of the system, from low-level hardware exceptions and interrupts, to context switches between concurrent processes, to abrupt changes in control flow caused by the delivery of Unix signals, to the nonlocal jumps in C that break the stack discipline.

  This is the part of the book where we introduce students to the fundamental idea of a process. Students learn how processes work and how they can be created and manipulated from application programs. We show them how application programmers can make use of multiple processes via Unix system calls. When students finish this chapter, they are able to write a Unix shell with job control.

- *Chapter 9: Measuring Program Execution Time.* This chapter teaches students how computers keep track of time (interval timers, cycle timers, and system clocks), the sources of error when we try to use these times to measure running time, and how to exploit this knowledge to get accurate measurements. To the best of our knowledge, this is unique material that has never been discussed before in any consistent way. We include the topic at this point because it requires an understanding of assembly language, processes, and caches.

- *Chapter 10: Virtual Memory.* Our presentation of the virtual memory system seeks to give students some understanding of how it works and its characteristics. We want students to know how it is that the different simultaneous processes can each use an identical range of addresses, sharing some pages but having individual copies of others. We also cover issues involved in managing and manipulating virtual memory. In particular, we cover the operation of storage allocators such as the Unix `malloc` and `free` operations. Covering this material serves several purposes. It reinforces the concept that the virtual memory space is just an array of bytes that the program can subdivide into different storage units. It helps students understand the effects of programs containing memory referencing errors such as storage leaks and invalid pointer references. Finally, many application programmers write their own storage allocators optimized toward the needs and characteristics of the application.

- *Chapter 11: System-Level I/O.* We cover the basic concepts of Unix I/O such as files and descriptors. We describe how files are shared, how I/O redirection works, and how to access file metadata. We also develop a robust buffered I/O package that deals correctly with short counts. We cover the C standard I/O library and its relationship to Unix I/O, focusing on limitations of standard I/O that make it unsuitable for network programming. In general, the topics covered in this chapter are building blocks for the next two chapters on network and concurrent programming.

- *Chapter 12: Network Programming.* Networks are interesting I/O devices to program, tying together many of the ideas that we have studied earlier in the text, such as processes, signals, byte ordering, memory mapping, and dynamic storage allocation. Network programs also provide a compelling context for concurrency, which is the topic of the next section. This chapter is a thin slice through network programming that gets the students to point where they can write a Web server. We cover the client-server model that underlies all network applications. We present a programmer's view of the Internet, and show students how to write Internet clients and servers using the sockets interface. Finally, we introduce HTTP and develop a simple iterative Web server.

- *Chapter 13: Concurrent Programming.* This chapter introduces students to concurrent programming using Internet server design as the running motivational example. We compare and contrast the three basic mechanisms for writing concurrent programs — processes, I/O multiplexing, and threads — and show how to use them to build concurrent Internet servers. We cover basic principles of synchronization using P and V semaphore operations, thread safety and reentrancy, race conditions, and deadlocks.

Courses Based on the Book

Instructors can use the CS:APP book to teach five different kinds of systems courses (Figure 2). The particular course depends on curriculum requirements, personal taste, and the backgrounds and abilities of the students. From left to right in the figure, the courses are characterized by an increasing emphasis on the programmer's perspective of a system. Here is a brief description:

- **ORG**: A computer organization course with traditional topics covered in an untraditional style. Traditional topics such as logic design, processor architecture, assembly language, and memory systems are covered. However, there is more emphasis on the impact for the programmer. For example, data representations are related back to their impact on C programs. Students learn how C constructs are represented in machine language.

- **ORG+**: The ORG course with additional emphasis on the impact of hardware on the performance of application programs. Compared to ORG, students learn more about code optimization and about improving the memory performance of their C programs.

- **ICS**: The baseline ICS course, designed to produce enlightened programmers who understand the impact of the hardware, operating system, and compilation system on the performance and correctness of their application programs. A significant difference from ORG+ is that low-level processor architecture is not covered. Instead, programmers work with a higher-level model of a modern out-of-order processor. The ICS course fits nicely into a 10-week quarter, and can also be stretched to a 15-week semester if covered at a more leisurely pace.

- **ICS+**: The baseline ICS course with additional coverage of systems programming topics such as system-level I/O, network programming, and concurrent programming. This is the semester-long Carnegie Mellon course, which covers every chapter in CS:APP except low-level processor architecture.

- **SP**: A systems programming course. Similar to the ICS+ course, but drops floating point and performance optimization, and places more emphasis on systems programming, including process control, dynamic linking, system-level I/O, network programming, and concurrent programming. Instructors might want to supplement from other sources for advanced topics such as daemons, terminal control, and Unix IPC.

		Course				
Chapter	Topic	ORG	ORG+	ICS	ICS+	SP
1	Tour of systems	●	●	●	●	●
2	Data representation	●	●	●	●	⊙ (d)
3	Machine language	●	●	●	●	●
4	Processor architecture	●	●			
5	Code optimization		●	●	●	
6	Memory hierarchy	⊙ (a)	●	●	●	⊙ (a)
7	Linking			⊙ (c)	⊙ (c)	●
8	Exceptional control flow			●	●	●
9	Performance measurement				●	●
10	Virtual memory	⊙ (b)	●	●	●	●
11	System-level I/O				●	●
12	Network programming				●	●
13	Concurrent programming				●	●

Figure 2: **Five systems courses based on the CS:APP book.** Notes: (a) Hardware only, (b) No dynamic storage allocation, (c) No dynamic linking, (d) No floating point. ICS+ is the 15-213 course from Carnegie Mellon.

The main message of Figure 2 is that the CS:APP book gives you a lot of options. If you want your students to be exposed to lower-level processor architecture, then that option is available via the ORG and ORG+ courses. On the other hand, if you want to switch from your current computer organization course to an ICS or ICS+ course, but are wary are making such a drastic change all at once, then you can move towards ICS incrementally. You can start with ORG, which teaches the traditional topics in an non-traditional way. Once you are comfortable with that material, then you can move to ORG+, and eventually to ICS. If students have no experience in C (for example they have only programmed in Java), you could spend several weeks on C and then cover the material of ORG or ICS.

Finally, we note that the ORG+ and SP courses would make a nice two-term (either quarters or semesters) sequence. Or you might consider offering ICS+ as one term of ICS and one term of SP.

Classroom-Tested Laboratory Exercises

The ICS+ course at Carnegie Mellon receives very high evaluations from students. Median scores of $5.0/5.0$ and means of $4.6/5.0$ are typical. Students cite the fun, exciting, and relevant laboratory exercises as the primary reason. Here are examples of the labs that are provided with the book:

- *Data Lab.* This lab requires students to implement simple logical and arithmetic functions, but using a highly restricted subset of C. For example, they must compute the absolute value of a number using only bit-level operations. This lab helps students understand the bit-level representations of C data types and the bit-level behavior of the operations on data.
- *Binary Bomb Lab.* A *binary bomb* is a program provided to students as an object code file. When run, it prompts the user to type in 6 different strings. If any of these is incorrect, the bomb "explodes," printing an error message and logging the event on a grading server. Students must "defuse" their own unique bomb by disassembling and reverse engineering the program to determine what the 6 strings should be. The lab teaches students to understand assembly language, and also forces them to learn how to use a debugger.
- *Buffer Overflow Lab.* Students are required to modify the run-time behavior of a binary executable by exploiting a buffer overflow bug. This lab teaches the students about the stack discipline and teaches them about the danger of writing code that is vulnerable to buffer overflow attacks.
- *Architecture Lab.* Several of the homework problems of Chapter 4 could be combined into a lab assignment, where students modify the HCL description of a processor to add new instructions, change the branch prediction policy, or add or remove bypassing paths and register ports. The resulting processors can be simulated and run through automated tests that will detect most of the possible bugs. This lab lets students experience the exciting parts of processor design without learning and constructing complex, low-level models in a language such as Verilog or VHDL.
- *Performance Lab.* Students must optimize the performance of an application kernel function such as convolution or matrix transposition. This lab provides a very clear demonstration of the properties of cache memories and gives them experience with low-level program optimization.
- *Shell Lab.* Students implement their own Unix shell program with job control, including the `ctrl-c` and `ctrl-z` keystrokes, `fg`, `bg`, and `jobs` commands. This is the student's first introduction to concurrency, and gives them a clear idea of Unix process control, signals, and signal handling.
- *Malloc Lab.* Students implement their own version of `malloc`, `free`, and (optionally) `realloc`. This lab gives students a clear understanding of data layout and organization, and requires them to evaluate different trade-offs between space and time efficiency.

- *Proxy Lab.* Students implement a concurrent Web proxy that sits between their browser and the rest of the World Wide Web. This lab exposes the students to such topics as web clients and servers, and ties together many of the concepts from the course, such as byte ordering, file I/O, process control, signals, signal handling, memory mapping, sockets, and concurrency.

The labs are available from the CS:APP Web page.

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.

Guy Blelloch, Greg Kesden, 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 on the impact of ICS on the OS course. 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. James Hoe provided useful ideas and feedback on how to present processor architecture.

A special group of students, Khalil Amiri, Angela Demke Brown, Chris Colohan, Jason Crawford, Peter Dinda, Julio Lopez, Bruce Lowekamp, Jeff Pierce, Sanjay Rao, Balaji Sarpeshkar, Blake Scholl, Sanjit Seshia, Greg Steffan, Tiankai Tu, Kip Walker, and Yinglian Xie 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, 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 did not 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), Wei Hsu (University of Minnesota), 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!

We would also like to thank our colleagues at Prentice Hall. Marcia Horton, Eric Frank, and Harold Stone have been unflagging in their support and vision. Jerry Ralya provided sharp insights and taught us a lot about good writing.

Finally, we would like to acknowledge the great technical writers Brian Kernighan and the late W. Richard Stevens, for showing us that technical books can be beautiful.

Thank you all.

Randy Bryant
Dave O'Hallaron

Pittsburgh, PA
February 1, 2002