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

You are poised for an exciting journey. If you dedicate yourself to learning the concepts in this book, then you will be on your way to becoming a rare "power programmer," enlightened by an understanding of the underlying computer system and its impact on your application programs.

You are going to learn practical skills such as how to avoid strange numerical errors caused by the way that computers represent numbers. You will learn how to optimize your C code by using clever tricks that exploit the designs of modern processors and memory systems. You will learn how the compiler implements procedure calls and how to use this knowledge to avoid the security holes from buffer overflow vulnerabilities that plague network and Internet software. You will learn how to recognize and avoid the nasty errors during linking that confound the average programmer. You will learn how to write your own Unix shell, your own dynamic storage allocation package, and even your own Web server. You will learn the promises and pitfalls of concurrency, a topic of increasing importance as multiple processor cores are integrated onto single chips.

In their classic text on the C programming language [61], Kernighan and Ritchie introduce readers to C using the `hello` program shown in Figure 1.1. 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 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.

code/intro/hello.c

```
1  #include <stdio.h>
2
3  int main()
4  {
5      printf("hello, world\n");
6      return 0;
7  }
```

code/intro/hello.c

Figure 1.1 The `hello` program. (Source: [60])

이 책은 애플리케이션 프로그램을 실행하는 데 필요한 시스템 구현에 대한 특정한 내용은 시간이 흐르면 변하지만 근본적인 개념은 변하지 않는다. 모든 컴퓨터 시스템은 유사한 기능을 수행하는 유사한 하드웨어 및 소프트웨어 구성 요소를 갖고 있다. 이 책은 이러한 구성 요소가 어떻게 작동하며 프로그램의 정확성과 성능에 어떤 영향을 미치는지 이해함으로써 자신의 기술을 향상시키고자 하는 프로그래머를 위해 쓰여졌다. 흥미진진한 여정을 앞두고 있습니다. 이 책의 개념을 배우는 데 헌신한다면, 희귀한 "파워 프로그래머"가 되기 위한 길로 나아가게 될 것입니다. 언덕별 컴퓨터 시스템과 애플리케이션 프로그램에 미치는 영향을 이해하는 것으로 깨달음을 얻을 수 있을 것입니다. 컴퓨터가 숫자를 표현하는 방식으로 인한 이상한 숫자 오류를 피하는 방법과 같은 실용적 기술을 배울 것입니다. 모던 프로세서 및 메모리 시스템의 설계를 이용하는 영리한 트릭을 사용하여 C 코드를 최적화하는 방법을 배울 것입니다. 컴파일러가 프로시저 호출을 어떻게 구현하는지 및 이러한 지식을 활용하여 네트워크 및 인터넷 소프트웨어를 괴롭히는 버퍼 오버플로 취약점을 피하는 방법을 배울 것입니다. 평범한 프로그래머를 괴롭히는 링킹 중 발생하는 불쾌한 오류를 인식하고 피하는 방법을 배울 것입니다. 자신만의 Unix 셸, 동적 저장 할당 패키지, 심지어 자체 웹 서버를 작성하는 방법을 배울 것입니다. 단일 칩에 여러 프로세서 코어가 통합되는 중요한 주제로서의 동시성의 약속과 위험을 배울 것입니다. C 프로그래밍 언어에 대한 고전적인 텍스트 [61]에서 Kernighan 및 Ritchie는 Figure 1.1에 나와 있는 hello 프로그램을 사용하여 C에 대한 독자를 소개합니다. hello는 매우 간단한 프로그램이지만, 완료되기 위해 모든 주요 부분을 함께 동작해야 합니다. 이 책의 목표는 당신의 시스템에서 hello를 실행했을 때 무엇이 일어나는지 이해하는 데 도움을 주는 것입니다. 우리는 프로그래머에 의해 생성되어 시스템에서 실행되고 간단한 메시지를 출력한 뒤 종료되는 hello 프로그램의 수명을 추적하며 해당 프로그램의 수명을 따를 때 사용되는 주요 개념, 용어 및 구성 요소를 간략히 소개할 것입니다. 이후 장에서는 이러한 아이디어에 대해 더 확장할 것입니다.

#	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	*SP*
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	32
SP	*SP*	*SP*	r	e	t	u	r	n	*SP*	0	;	\n	}	\n	
32	32	32	114	101	116	117	114	110	32	48	59	10	125	10	

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

1.1 Information Is Bits + 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 computer systems represent text characters using the ASCII standard that represents each character with a unique byte-size integer value.[1] For example, Figure 1.2 shows the ASCII representation of the `hello.c` program.

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.

As programmers, we need to understand machine representations of numbers because they are not the same as integers and real numbers. They are finite

1. Other encoding methods are used to represent text in non-English languages. See the aside on page 86 for a discussion on this.

우리의 hello 프로그램은 소스 프로그램(또는 소스 파일)으로 시작합니다. 프로그래머가 편집기로 만들고 hello.c 라는 텍스트 파일에 저장합니다. 소스 프로그램은 0 또는 1의 값을 갖는 비트(각각 8비트의 바이트로 구성된)의 연속입니다. 각 바이트는 프로그램의 텍스트 문자를 나타냅니다. 대부분의 컴퓨터 시스템은 ASCII 표준을 사용하여 텍스트 문자를 나타냅니다. 이 표준은 각 문자를 고유한 바이트 크기의 정수 값으로 표현합니다. 예를 들어, 그림 1.2는 hello.c 프로그램의 ASCII 표현을 보여줍니다. hello.c 프로그램은 바이트 시퀀스로 파일에 저장됩니다. 각 바이트에는 문자에 해당하는 정수 값이 있습니다. 예를 들어, 첫 번째 바이트의 정수 값은 35로, 이는 문자 '#'에 해당합니다. 두 번째 바이트의 정수 값은 105로, 이는 문자 'i'에 해당합니다. 이와 같이, 각 텍스트 라인은 보이지 않는 새 줄 문자 '₩n'으로 끝남을 주목해야 합니다. 이는 정수 값 10으로 표현됩니다. ASCII 문자로만 구성된 hello.c와 같은 파일들은 텍스트 파일로 알려져 있습니다. 다른 파일들은 바이너리 파일로 알려져 있습니다. hello.c의 표현은 기본 개념을 보여줍니다. 시스템의 모든 정보는 디스크 파일, 메모리에 저장된 프로그램, 메모리에 저장된 사용자 데이터, 그리고 네트워크를 통해 전송된 데이터를 비롯해 비트들의 집합으로 표현됩니다. 다른 데이터 객체를 구별하는 유일한 것은 우리가 그것들을 보는 문맥입니다. 예를 들어, 다른 문맥에서 동일한 바이트 시퀀스는 정수, 부동 소숫점 수, 문자열 또는 기계 명령어를 나타낼 수 있습니다. 프로그래머로서, 우리는 숫자의 기계적인 표현을 이해해야 합니다. 왜냐하면 그것들은 정수와 실수와 같지 않기 때문입니다. 그것들은 유한합니다.

다른 인코딩 방법은 영어 이외의 언어로 된 텍스트를 표현하는 데 사용됩니다. 이에 대한 토론은 페이지 86의 쪽지를 참조하십시오.

Aside Origins of the C programming language

C was developed from 1969 to 1973 by Dennis Ritchie of Bell Laboratories. The American National Standards Institute (ANSI) ratified the ANSI C standard in 1989, and this standardization later became the responsibility of the International Standards Organization (ISO). The standards define 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" [61]. In Ritchie's words [92], 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 (the core part of the operating system), 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 base 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, objects, and exceptions. Newer languages such as C++ and Java address these issues for application-level programs.

C는 1969년부터 1973년까지 벨 연구소의 데니스 리치에에 의해 개발되었습니다. 미국 국가 표준협회(ANSI)는 1989년 ANSI C 표준을 승인했으며, 이 표준화는 나중에 국제 표준 기구(ISO)의 책임이 되었습니다. 이 표준은 C 언어와 C 표준 라이브러리로 알려진 일련의 라이브러리 함수를 정의합니다. 커니한과 리치는 그들의 고전적인 책에서 ANSI C를 설명하며, 이 책은 애착으로 "K&R;"이라고 불립니다 [61]. 리치의 말에 따르면 [92], C는 "별난, 결함이 있지만 엄청난 성공"입니다. 그래서 왜 이렇게 성공한 것일까요?

C는 Unix 운영 체제와 밀접한 관련이 있었습니다. C는 처음부터 Unix를 위한 시스템 프로그래밍 언어로 개발되었습니다. Unix의 대부분 커널(운영 체제의 핵심 부분) 및 모든 지원 도구와 라이브러리는 C로 작성되었습니다. Unix가 1970년대 후반과 1980년대 초반에 대학에서 인기를 얻으면서 많은 사람들이 C를 접하고 마음에 들었다는 것을 발견했습니다. Unix가 거의 완전히 C로 작성되어 있었기 때문에 새로운 기계에 쉽게 이식될 수 있어, C와 Unix에게 더욱 넓은 관객이 생겼습니다.

C 언어는 작고 간단한 언어입니다. 이 설계는 위원회가 아니라 한 사람에 의해 통제되었으며 결과물은 깨끗하고 일관된 설계로서 불필요한 부담이 거의 없었습니다. K&R; 책은 단 261페이지로 완전한 언어와 표준 라이브러리를 다양한 예제와 연습문제와 함께 설명합니다. C의 간결함은 상대적으로 쉽게 배울 수 있고 다른 컴퓨터로 이식할 수 있게 만들었습니다.

C 언어는 실용적인 목적으로 설계되었습니다. C 언어는 유닉스 운영 체제를 구현하기 위해 설계되었습니다. 나중에 다른 사람들은 언어에 방해받지 않고 원하는 프로그램을 작성할 수 있다는 것을 발견했습니다.

C 언어는 시스템 수준 프로그래밍을 위한 선택 언어이며, 응용 프로그램 수준의 프로그램도 광범위하게 사용됩니다. 그러나 모든 프로그래머와 상황에 완벽하지는 않습니다. C 포인터는 혼란과 프로그래밍 오류의 일반적인 원인입니다. 또한 C는 클래스, 객체 및 예외와 같은 유용한 추상화를 명시적으로 지원하지 않습니다. C++ 및 Java와 같은 최신 언어는 이러한 문제를 응용 프로그램 수준에서 해결합니다.

approximations that can behave in unexpected ways. This fundamental idea is explored in detail in Chapter 2.

예상치 못한 방식으로 동작할 수 있는 근사치입니다. 이 기본적인 아이디어는 2장에서 자세히 탐구됩니다.

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:*

"hello 프로그램은 고수준 C 프로그램으로 시작되며, 그 형태로 사람들이 읽고 이해할 수 있습니다. 그러나 시스템에서 hello.c를 실행하기 위해 개별 C 문은 다른 프로그램에 의해 저수준 기계어 명령어의 순서로 번역되어야 합니다. 이러한 명령어들은 그 후에 실행 가능한 객체 프로그램이라는 형태로 속이고, 바이너리 디스크 파일로 저장됩니다. 객체 프로그램은 실행 가능한 객체 파일로도 불립니다."