C Programming

eC

C.1 INTRODUCTION

The overall goal of this book is to give a picture of how computers work on many levels, from the transistors by which they are constructed all the way up to the software they run. The first five chapters of this book work up through the lower levels of abstraction, from transistors to gates to logic design. Chapters 6 through 8 jump up to architecture and work back down to microarchitecture to connect the hardware with the software. This appendix on C programming fits logically between Chapters 5 and 6, covering C programming as the highest level of abstraction in the text. It motivates the architecture material and links this book to programming experience that may already be familiar to the reader. This material is placed in the appendix so that readers may easily cover or skip it depending on previous experience.

Programmers use many different languages to tell a computer what to do. Fundamentally, computers process instructions in *machine language* consisting of 1's and 0's, as is explored in Chapter 6. But programming in machine language is tedious and slow, leading programmers to use more abstract languages to get their meaning across more efficiently. Table eC.1 lists some examples of languages at various levels of abstraction.

One of the most popular programming languages ever developed is called C. It was created by a group including Dennis Ritchie and Brian Kernighan at Bell Laboratories between 1969 and 1973 to rewrite the UNIX operating system from its original assembly language. By many measures, C (including a family of closely related languages such as C++, C#, and Objective C) is the most widely used language in existence. Its popularity stems from a number of factors, including its:

▸ Availability on a tremendous variety of platforms, from supercomputers down to embedded microcontrollers

▸ Relative ease of use, with a huge user base

Table eC.1 Languages at roughly decreasing levels of abstraction

Language	Description
MATLAB	Designed to facilitate heavy use of math functions
Perl	Designed for scripting
Python	Designed to emphasize code readability
Java	Designed to run securely on any computer
C	Designed for flexibility and overall system access, including device drivers
Assembly Language	Human-readable machine language
Machine Language	Binary representation of a program

Dennis Ritchie, 1941–2011

Brian Kernighan, 1942–
C was formally introduced in 1978 by Brian Kernighan and Dennis Ritchie's classic book, *The C Programming Language*. In 1989, the American National Standards Institute (ANSI) expanded and standardized the language, which became known as ANSI C, Standard C, or C89. Shortly thereafter, in 1990, this standard was adopted by the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). ISO/IEC updated the standard in 1999 to what is called C99, which we will be discussing in this text.

▶ Moderate level of abstraction, providing higher productivity than assembly language, yet giving the programmer a good understanding of how the code will be executed
▶ Suitability for generating high-performance programs
▶ Ability to interact directly with the hardware

This chapter is devoted to C programming for a variety of reasons. Most importantly, C allows the programmer to directly access addresses in memory, illustrating the connection between hardware and software emphasized in this book. C is a practical language that all engineers and computer scientists should know. Its uses in many aspects of implementation and design—for example, software development, embedded systems programming, and simulation—make proficiency in C a vital and marketable skill.

The following sections describe the overall syntax of a C program, discussing each part of the program—including the header, function and variable declarations, data types, and commonly used functions provided in libraries. Chapter 9 (available as a web supplement— see Preface) describes a hands-on application by using C to program SparkFun's RED-V RedBoard, which contains a RISC-V microcontroller.

SUMMARY

▶ **High-level programming:** High-level programming is useful at many levels of design, from writing analysis or simulation software to programming microcontrollers that interact with hardware.
▶ **Low-level access:** C code is powerful because, in addition to high-level constructs, it provides access to low-level hardware and memory.

C.2 WELCOME TO C

A C program is a text file that describes operations for the computer to perform. The text file is *compiled*, converted into a machine-readable format, and run or *executed* on a computer. C Code Example eC.1 is a simple C program that prints the phrase "Hello world!" to the *console*, the computer screen. C programs are generally contained in one or more text files that end in ".c." Good programming style requires a file name that indicates the contents of the program—for example, this file could be called hello.c.

C Code Example eC.1 SIMPLE C PROGRAM

```
// Write "Hello world!" to the console
#include <stdio.h>

int main(void) {
  printf("Hello world!\n");
}
```

Console Output

```
Hello world!
```

C.2.1 C Program Dissection

In general, a C program is organized into one or more functions. Every program must include the main function, which is where the program starts executing. Most programs use other functions defined elsewhere in the C code and/or in a library. The overall sections of the hello.c program are the header, the main function, and the body.

Header: #include <stdio.h>
The header includes the *library functions* needed by the program. In this case, the program uses the printf function, which is part of the standard I/O library, stdio.h. See Section C.9 for further details on C's built-in libraries.

Main function: int main(void)
All C programs must include exactly one main function. Execution of the program occurs by running the code inside main, called the *body* of main. Function syntax is described in Section C.6. The body of a function contains a sequence of *statements*. Each statement ends with a semicolon. int denotes that the main function outputs, or *returns*, an integer result that indicates whether the program ran successfully.

C is the language used to program such ubiquitous systems as Linux, Windows, and iOS. C is a powerful language because of its direct access to hardware. As compared with other high-level languages—for example, Perl and MATLAB—C does not have as much built-in support for specialized operations such as file manipulation, pattern matching, matrix manipulation, and graphical user interfaces. It also lacks features to protect the programmer from common mistakes, such as writing data past the end of an array. Its power combined with its lack of protection has assisted hackers who exploit poorly written software to break into computer systems.

While this chapter provides a fundamental understanding of C programming, entire texts are written that describe C in depth. One of our favorites is the classic text *The C Programming Language* by Brian Kernighan and Dennis Ritchie, the developers of C. This text gives a concise description of the nuts and bolts of C. Another good text is *A Book on C* by Al Kelley and Ira Pohl.

Body: `printf("Hello world!\n");`

The body of this `main` function contains one statement, a call to the `printf` function, which prints the phrase "Hello world!" followed by a newline character indicated by the special sequence "\n". Further details about I/O (input/output) functions are described in Section C.9.1.

All programs follow the general format of the simple `hello.c` program. Of course, very complex programs may contain millions of lines of code and span hundreds of files.

C.2.2 Running a C Program

C programs can be run on many different machines. This *portability* is another advantage of C. The program is first compiled on the desired machine using the *C compiler*. Slightly different versions of the C compiler exist, including *cc* (C compiler), or *gcc* (GNU C compiler). Here, we show how to compile and run a C program using gcc, which is freely available for download. It runs directly on Linux machines and is accessible under the Cygwin environment on Windows machines. It is also available for many embedded systems, such as SparkFun's RED-V RedBoard, which includes a RISC-V microcontroller. The general process described below of C file creation, compilation, and execution is the same for any C program.

1. Create the text file, for example, `hello.c`.
2. In a terminal window, change to the directory that contains the file `hello.c` and type `gcc hello.c` at the command prompt.
3. The compiler creates an executable file. By default, the executable is called `a.out` (or `a.exe` on Windows machines).
4. At the command prompt, type `./a.out` (or `./a.exe` on Windows) and press return.
5. "Hello world!" will appear on the screen.

SUMMARY

▶   `filename.c`: C program files are typically named with a .c extension.

▶   `main`: Each C program must have exactly one `main` function.

▶   `#include`: Most C programs use functions provided by built-in libraries. These functions are used by writing `#include <library.h>` at the top of the C file.

▶   `gcc filename.c`: C files are converted into an executable using a compiler such as the GNU compiler (gcc) or the C compiler (cc).

▶   **Execution:** After compilation, C programs are executed by typing `./a.out` (or `./a.exe`) at the command line prompt.

C.3 COMPILATION

A compiler is a piece of software that reads a program in a high-level language and converts it into a file of machine code called an executable. Entire textbooks are written on compilers, but we describe them here briefly. The overall operation of the compiler is to (1) preprocess the file by including referenced libraries and expanding macro definitions, (2) ignore all unnecessary information such as comments, (3) translate the high-level code into simple instructions native to the processor that are represented in binary, called machine language, and (4) compile all the instructions into a single binary executable that can be read and executed by the computer. Each machine language is specific to a given processor, so a program must be compiled specifically for the system on which it will run. For example, the RISC-V machine language is covered in Chapter 6 in detail.

C.3.1 Comments

Programmers use comments to describe code at a high level and clarify code function. Anyone who has read uncommented code can attest to its importance. C programs use two types of comments: Single-line comments begin with // and terminate at the end of the line; multiple-line comments begin with /* and end with */. While comments are critical to the organization and clarity of a program, they are ignored by the compiler.

```
// This is an example of a one-line comment.
/* This is an example
   of a multi-line comment. */
```

A comment at the top of each C file is useful to describe the file's author, creation and modification dates, and purpose. The comment below could be included at the top of the hello.c file.

```
// hello.c
// 15 Jan 2021 Sarah.Harris@unlv.edu, David_Harris@hmc.edu
//
// This program prints "Hello world!" to the screen
```

C.3.2 #define

Constants are named using the #define directive and then used by name throughout the program. These globally defined constants are also called *macros*. For example, suppose you write a program that allows at most 5 user guesses. You can use #define to identify that number.

```
#define MAXGUESSES 5
```

The # indicates that this line in the program will be handled by the *preprocessor*. Before compilation, the preprocessor replaces each occurrence

Number constants in C default to decimal but can also be hexadecimal (prefix `"0x"`) or octal (prefix `"0"`)[1]. Binary constants are not defined in C99 but are supported by some compilers (prefix `"0b"`). For example, the following assignments are equivalent:

```
char x = 37;
char x = 0x25;
char x = 045;
char x = 0b100101;
```

Globally defined constants eradicate *magic numbers* from a program. A magic number is a constant that shows up in a program without a name. The presence of magic numbers in a program often introduces tricky bugs—for example, when the number is changed in one location but not another.

of the identifier `MAXGUESSES` in the program with `5`. By convention, `#define` lines are located at the top of the file and identifiers are written in all capital letters. By defining constants in one location and then using the identifier in the program, the program remains consistent, and the value is easily modified—it need only be changed at the `#define` line instead of at each line in the code where the value is used.

C Code Example eC.2 shows how to use the `#define` directive to convert inches to centimeters. The variables `inch` and `cm` are declared to be `float`, which means they represent single-precision floating-point numbers. If the conversion factor (`INCH2CM`) were used throughout a large program, having it declared using `#define` obviates errors due to typos (e.g., typing 2.53 instead of 2.54) and makes it easy to find and change (e.g., if more significant digits were required).

C Code Example eC.2 USING `#define` TO DECLARE CONSTANTS

```
// Convert inches to centimeters
#include <stdio.h>
#define INCH2CM 2.54

int main(void) {
  float inch = 5.5;     // 5.5 inches
  float cm;

  cm = inch * INCH2CM;
  printf("%f inches = %f cm\n", inch, cm);
}
```

Console Output

```
5.500000 inches = 13.970000 cm
```

C.3.3 `#include`

Modularity encourages us to split programs across separate files and functions. Commonly used functions can be grouped together for easy reuse. Variable declarations, defined values, and function definitions located in a *header file* can be used by another file by adding the `#include` preprocesser directive. *Standard libraries* that provide commonly used functions are accessed in this way. For example, the following line is required to use the functions defined in the standard input/output (I/O) library, such as `printf`.

```
#include <stdio.h>
```

The ".h" postfix of the `include` file indicates that it is a header file. While `#include` directives can be placed anywhere in the file before the included

---

[1]The prefixes for hexadecimal, octal, and binary (`0x`, `0`, and `0b`) start with the number 0, not the letter O.

functions, variables, or identifiers are needed, they are conventionally located at the top of a C file.

Programmer-created header files can also be included but must use quotation marks (" ") around the file name instead of brackets (< >). For example, a user-created header file called `myfunctions.h` would be included using the following line.

```
#include "myfunctions.h"
```

At compile time, files specified in brackets are searched for in system directories. Files specified in quotes are searched for in the same local directory where the C file is found. If the user-created header file is located in a different directory, the path of the file relative to the current directory must be included.

SUMMARY

▸ **Comments:** C provides single-line comments (`//`) and multi-line comments (`/* */`).

▸ `#define NAME val`: the `#define` directive allows an identifier (`NAME`) to be used throughout the program. Before compilation, all instances of `NAME` are replaced with `val`.

▸ `#include`: `#include` allows common functions to be used in a program. For built-in libraries, include the following line at the top of the code: `#include <library.h>`. To include a user-defined header file, the name must be in quotes, listing the path relative to the current directory as needed: that is, `#include "other/myFuncs.h"`.

C.4 VARIABLES

Variables in C programs have a type, name, value, and memory location. A variable declaration states the type and name of the variable. For example, the following declaration states that the variable is of type `char` (which is a 1-byte type), and that the variable name is `x`. The compiler decides where to place this 1-byte variable in memory.

```
char x;
```

C views memory as a group of consecutive bytes, where each byte of memory is assigned a unique number indicating its location or *address*, as shown in Figure eC.1. A variable occupies one or more bytes of memory; the address of multiple-byte variables is indicated by the lowest numbered byte. The type of a variable indicates whether to interpret the byte(s) as an integer, floating-point number, or other type. The rest of this section describes C's primitive data types, the declaration of global and local variables, and the initialization of variables.

> Variable names are case sensitive and can be of your choosing. However, the name may not be any of C's reserved words (i.e., `int`, `while`, etc.), may not start with a number (i.e., `int 1x;` is not a valid declaration), and may not include special characters such as `\`, `*`, `?`, or `-`. Underscores (`_`) are allowed.
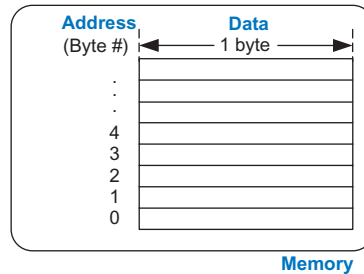
C.4.1 Primitive Data Types

C has a number of primitive, or built-in, data types available. They can be broadly characterized as integers, floating-point variables, and characters. An integer represents a two's complement or unsigned number within a finite range. A floating-point variable uses IEEE floating-point representation to describe real numbers with a finite range and precision. A character can be viewed as either an ASCII value or an 8-bit integer.[2] Table eC.2 lists the size and range of each primitive type. Integers may be 16, 32, or 64 bits. They use two's complement unless qualified as unsigned.

Table eC.2 Primitive data types and sizes

Type	Size (bits)	Minimum	Maximum
char	8	$-2^{-7} = -128$	$2^7 - 1 = 127$
unsigned char	8	0	$2^8 - 1 = 255$
int	machine-dependent		
unsigned int	machine-dependent		
int16_t	16	$-2^{15} = -32,768$	$2^{15} - 1 = 32,767$
uint16_t	16	0	$2^{16} - 1 = 65,535$
int32_t	32	$-2^{31} = -2,147,483,648$	$2^{31} - 1 = 2,147,483,647$
uint32_t	32	0	$2^{32} - 1 = 4,294,967,295$
int64_t	64	-2^{63}	$2^{63} - 1$
uint64_t	64	0	$2^{64} - 1$
float	32	$\pm 2^{-126}$	$\pm 2^{127}$
double	64	$\pm 2^{-1023}$	$\pm 2^{1022}$

---

[2] Technically, the C99 standard defines a character as "a bit representation that fits in a byte," without requiring a byte to be 8 bits. However, current systems define a byte as 8 bits.
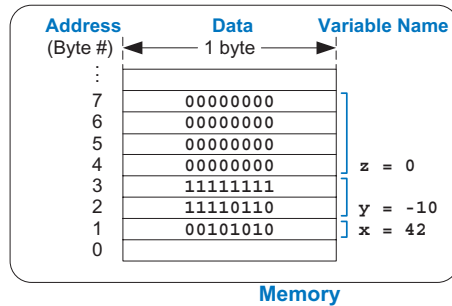
The size of the `int` type is machine dependent and is often the native word size of the machine. For example, on a 32-bit processor, the size of an `int` or `unsigned int` is 32 bits. On a 16-bit processor, an `int` is usually 16 bits. However, compilers for 64-bit processors typically use 32 bits for `int`s to reduce subtle bugs porting old code that assumed this size. If you care about the size of a data type, use `int16_t`, `int32_t`, or `int64_t` to explicitly define the size. (These are signed datatypes; their unsigned counterparts are `uint16_t`, etc.) Floating-point numbers use 32 or 64 bits for single- or double-precision, respectively. Characters are 8 bits.

C Code Example eC.3 shows the declaration of variables of different types. As shown in Figure eC.2, `x` requires one byte of data, `y` requires two, and `z` requires four. The program decides where these bytes are stored in memory, but each type always requires the same amount of data. For illustration, the addresses of `x`, `y`, and `z` in this example are 1, 2, and 4. Variable names are case-sensitive, so, for example, the variable `x` and the variable `X` are two different variables. (But it would be very confusing to use both in the same program!)

C Code Example eC.3 EXAMPLE DATA TYPES

```
// Examples of several data types and their binary representations
unsigned char x = 42;        // x = 00101010
int16_t y = −10;             // y = 11111111 11110110
unit32_t z = 2;              // z = 00000000 00000000 00000000 00000010
```

C.4.2 Global and Local Variables

Global and local variables differ in where they are declared and where they are visible. A global variable is declared outside of all functions, typically at the top of a program, and can be accessed by all functions. Global variables should be used sparingly because they violate the principle of modularity, making large programs more difficult to read. However, a variable accessed by many functions can be made global.

A local variable is declared inside a function and can only be used by that function. Therefore, two functions can have local variables with

The *scope* of a variable is the context in which it can be used. For example, for a local variable, its scope is the function in which it is declared. It is out of scope everywhere else.

the same names without interfering with each other. Local variables are declared at the beginning of a function. They cease to exist when the function ends and are recreated when the function is called again. They do not retain their value from one invocation of a function to the next.

C Code Examples eC.4 and eC.5 compare programs using global versus local variables. In C Code Example eC.4, the global variable max can be accessed by any function. Using a local variable, as shown in C Code Example eC.5, is the preferred style because it preserves the well-defined interface of modularity.

C Code Example eC.4 GLOBAL VARIABLES

```
// Use a global variable to find and print the maximum of 3 numbers
int max;                  // global variable holding the maximum value
void findMax(int a, int b, int c) {
   max = a;
   if (b > max) {
     if (c > b) max = c;
     else       max = b;
   } else if (c > max) max = c;
}

void printMax(void) {
   printf("The maximum number is: %d\n", max);
}

int main(void) {
   findMax(4, 3, 7);
   printMax();
}
```

C Code Example eC.5 LOCAL VARIABLES

```
// Use local variables to find and print the maximum of 3 numbers
int getMax(int a, int b, int c) {
   int result = a;  // local variable holding the maximum value

   if (b > result) {
     if (c > b) result = c;
     else       result = b;
   } else if (c > result) result = c;

   return result;

}
void printMax(int m) {
   printf("The maximum number is: %d\n", m);
}
int main(void) {
   int max;

   max = getMax(4, 3, 7);
   printMax(max);
}
```

C.4.3 Initializing Variables

A variable needs to be *initialized*—assigned a value—before it is read. When a variable is declared, the correct number of bytes is reserved for that variable in memory. However, the memory at those locations retains whatever value it had the last time it was used, essentially a random value. Global and local variables can be initialized either when they are declared or within the body of the program. C Code Example eC.3 shows variables initialized at the same time they are declared. C Code Example eC.4 shows how variables are initialized before their use, but after declaration; the global variable `max` is initialized by the `getMax` function before it is read by the `printMax` function. Reading from uninitialized variables is a common programming error and can be tricky to debug.

SUMMARY

- **Variables:** Each variable is defined by its data type, name, and memory location. A variable is declared as `data type name`.

- **Data types:** A data type describes the size (number of bytes) and representation (interpretation of the bytes) of a variable. Table eC.2 lists C's built-in data types.

- **Memory:** C views memory as a list of bytes. Memory stores variables and associates each variable with an address (byte number).

- **Global variables:** Global variables are declared outside of all functions and can be accessed anywhere in the program.

- **Local variables:** Local variables are declared within a function and can be accessed only within that function.

- **Variable initialization:** Each variable must be initialized before it is read. Initialization can happen either at declaration or afterward.

C.5 OPERATORS

The most common type of statement in a C program is an *expression*, such as

```
y = a + 3;
```

An expression involves operators (such as + or *) acting on one or more operands, such as variables or constants. C supports the operators shown in Table eC.3, listed by category and in order of decreasing precedence. For example, multiplicative operators take precedence over additive operators. Within the same category, operators are evaluated in the order that they appear in the program.

Table eC.3 Operators listed by decreasing precedence

Category	Operator	Description	Example
Unary	++	post-increment	`a++; // a = a+1`
	−−	post-decrement	`x--; // x = x−1`

	&	memory address of a variable	`x = &y; // x = the memory`
			`            // address of y`

	~	bitwise NOT	`z = ~a;`
	!	Boolean NOT	`!x`
	−	negation	`y = −a;`
	++	pre-increment	`++a; // a = a + 1`
	−−	pre-decrement	`--x; // x = x − 1`

	`(type)`	casts a variable to (type)	`x = (int)c; // cast c to an int and`
			`                // assign it to x`

	`sizeof()`	size of a variable or type in bytes	`int32_t y;` `x = sizeof(y); // x = 4`
Multiplicative	*	multiplication	`y = x * 12;`
	/	division	`z = 9 / 3; // z = 3`
	%	modulo	`z = 5 % 2; // z = 1`
Additive	+	addition	`y = a + 2;`
	−	subtraction	`y = a − 2;`
Bitwise Shift	<<	bitshift left	`z = 5 << 2; // z = 0b00010100`
	>>	bitshift right	`x = 9 >> 3; // x = 0b00000001`
Relational	==	equals	`y == 2`
	!=	not equals	`x != 7`
	<	less than	`y < 12`
	>	greater than	`val > max`
	<=	less than or equal	`z <= 2`
	>=	greater than or equal	`y >= 10`

Table eC.3 Operators listed by decreasing precedence—cont'd

Category	Operator	Description	Example
Bitwise	&	bitwise AND	`y = a & 15;`
	^	bitwise XOR	`y = 2 ^ 3;`
	|	bitwise OR	`y = a	b;`
Logical	&&	Boolean AND	`x && y`
	||	Boolean OR	`x || y`

Ternary	? :	ternary operator	`y = x ? a : b; //if x is TRUE,`
			`                 // y = a, else y = b`

Assignment	=	assignment	`x = 22;`
	+=	addition and assignment	`y += 3;        // y = y + 3`
	−=	subtraction and assignment	`z −= 10;       // z = z − 10`
	*=	multiplication and assignment	`x *= 4;        // x = x * 4`
	/=	division and assignment	`y /= 10;       // y = y / 10`
	%=	modulo and assignment	`x %= 4;        // x = x % 4`
	>>=	bitwise right-shift and assignment	`x >>= 5;       // x = x >> 5`
	<<=	bitwise left-shift and assignment	`x <<= 2;       // x = x << 2`
	&=	bitwise AND and assignment	`y &= 15;       // y = y & 15`
	|=	bitwise OR and assignment	`x |= y; // x = x	y`
	^=	bitwise XOR and assignment	`x ^= y;        // x = x ^ y`

The Truth, the Whole Truth, and Nothing But the Truth

C considers a variable to be TRUE if it is nonzero and FALSE if it is zero. Logical and ternary operators, as well as control-flow statements such as `if` and `while`, depend on the truth of a variable. Relational and logical operators produce a result that is 1 when TRUE or 0 when FALSE.

Unary operators, also called monadic operators, have a single operand. Ternary operators have three operands, and all others have two. The ternary operator (from the Latin *ternarius*, meaning consisting of three) chooses the second or third operand depending on whether the first value is TRUE (nonzero) or FALSE (zero), respectively. C Code Example eC.6 shows how to compute `y = max(a,b)` using the ternary operator, along with an equivalent but more verbose `if/else` statement.

C Code Example eC.6 (a) TERNARY OPERATOR, AND (b) EQUIVALENT
if/else STATEMENT

```
(a) y = (a > b) ? a : b; // parentheses not necessary, but makes it clearer
(b) if (a > b) y = a;
    else      y = b;
```

Simple assignment uses the = operator. C code also allows for compound assignment, that is, assignment after a simple operation such as addition (+=) or multiplication (*=). In compound assignments, the variable on the left side is both operated on and assigned the result. C Code Example eC.7 shows these and other C operations. Binary values in the comments are indicated with the prefix "0b."

C Code Example eC.7 OPERATOR EXAMPLES

Expression	Result	Notes
`53 / 14`	`3`	Integer division truncates
`53 % 14`	`11`	53 mod 14
`0x2C && 0xE   // 0b101100 && 0b1110`	`1`	Logical AND
`0x2C || 0xE   // 0b101100 || 0b1110`	`1`	Logical OR
`0x2C & 0xE    // 0b101100 & 0b1110`	`0xC  (0b001100)`	Bitwise AND
`0x2C | 0xE    // 0b101100 | 0b1110`	`0x2E (0b101110)`	Bitwise OR
`0x2C ^ 0xE    // 0b101100 ^ 0b1110`	`0x22 (0b100010)`	Bitwise XOR
`0xE << 2      // 0b1110 << 2`	`0x38 (0b111000)`	Left shift by 2
`0x2C >> 3     // 0b101100 >> 3`	`0x5  (0b101)`	Right shift by 3
`x = 14;` `x += 2;`	`x = 16`	

`y = 0x2C;     // y = 0b101100`
`y &= 0xF;      // y = y & 0b1111`
	`y = 0xC (0b001100)`	

`x = 14; y = 83;` `y = y + x++;`	`x = 15, y = 97`	Increment x after using it
`x = 14; y = 83;` `y = y + ++x;`	`x = 15, y = 98`	Increment x before using it

C.6 FUNCTION CALLS

Modularity is key to good programming. A large program is divided into smaller parts called functions that, similar to hardware modules, have well-defined inputs, outputs, and behavior. C Code Example eC.8 shows the `sum3` function. The function declaration begins with the return type, `int`, followed by the name, `sum3`, and the inputs enclosed within parentheses (`int a, int b, int c`). Curly braces `{}` enclose the body of the function, which may contain zero or more statements. The return statement indicates the value that the function should return to its caller; this can be viewed as the output of the function. A function can return only a single value.

Functions are also referred to as procedures.

C Code Example eC.8 `sum3` **FUNCTION**

```
// Return the sum of the three input variables
int sum3(int a, int b, int c) {
   int result = a + b + c;
   return result;
}
```

After the following call to `sum3`, `y` holds the value 42.

```
int y = sum3(10, 15, 17);
```

Although a function may have inputs and outputs, neither is required. C Code Example eC.9 shows a function with no inputs or outputs. The keyword `void` before the function name indicates that nothing is returned. `void` between the parentheses indicates that the function has no input arguments.

C Code Example eC.9 **FUNCTION** `printPrompt` **WITH NO INPUTS OR OUTPUTS**

```
// Print a prompt to the console
void printPrompt(void) {
   printf("Please enter a number from 1-3:\n");
}
```

Nothing between the parentheses also indicates no input arguments. So, in this case, we could have written:

```
void printPrompt()
```

A function must be declared in the code before it is called. This may be done by placing the called function earlier in the file. For this reason, `main` is often placed at the end of the C file after all of the functions it calls. Alternatively, a function *prototype* can be placed in the program before the function is defined. The function prototype is the first line

of the function; it declares the return type, function name, and function inputs. For example, the function prototypes for the functions in C Code Examples eC.8 and eC.9 are:

```
int sum3(int a, int b, int c);
void printPrompt(void);
```

C Code Example eC.10 shows how function prototypes are used. Even though the functions themselves are after main, the function prototypes at the top of the file allow them to be used in main.

C Code Example eC.10 FUNCTION PROTOTYPES

```
#include <stdio.h>
// function prototypes
int sum3(int a, int b, int c);
void printPrompt(void);

int main(void) {
  int y = sum3(10, 15, 20);
  printf("sum3 result: %d\n", y);
  printPrompt();
}

int sum3(int a, int b, int c) {
  int result = a + b + c;
  return result;
}

void printPrompt(void) {
  printf("Please enter a number from 1−3:\n");
}
```

Console Output

```
sum3 result: 45
Please enter a number from 1−3:
```

As with variable names, function names are case sensitive, cannot be any of C's reserved words, may not contain special characters (except underscore _), and cannot start with a number. Typically, function names include a verb to indicate what they do.

Be consistent in how you capitalize your function and variable names so you don't have to constantly look up the correct capitalization. Two common styles are to camelCase, in which the initial letter of each word after the first is capitalized like the humps of a camel (e.g., printPrompt), or to use underscores between words (e.g., print_prompt). We have unscientifically observed that reaching for the underscore key exacerbates carpal tunnel syndrome (my pinky finger twinges just thinking about the underscore!) and hence prefer camelCase. But the most important thing is to be consistent in style within your organization.

The main function is always declared to return an int; this return value tells the operating system the reason for program termination. A zero indicates normal completion, while a nonzero value signals an error condition. If main reaches the end without encountering a return statement, it will automatically return 0. Most operating systems do not automatically inform the user of the value returned by the program.

C.7 CONTROL-FLOW STATEMENTS

C provides control-flow statements for conditionals and loops. Conditionals execute a statement only if a condition is met. A loop repeatedly executes a statement as long as a condition is met.

C.7.1 Conditional Statements

If, if/else, and switch/case statements are conditional statements commonly used in high-level languages, including C.

If Statements

An if statement executes the statement immediately following it when the expression in parentheses is TRUE (i.e., nonzero). The general format is:

```
if (expression)
   statement
```

C Code Example eC.11 shows how to use an if statement in C. When the variable aintBroke is equal to 1, the variable dontFix is set to 1. A block of multiple statements can be executed by placing curly braces {} around the statements, as shown in C Code Example eC.12.

Curly braces, {}, are used to group zero or more statements into a *compound statement* or *block*.

C Code Example eC.11 IF STATEMENT

```
int dontFix = 0;
if (aintBroke == 1)
   dontFix = 1;
```

C Code Example eC.12 IF STATEMENT WITH A BLOCK OF STATEMENTS

```
// If amt >= $2, prompt user and dispense candy
if (amt >= 2) {
  printf("Select candy.\n");
  dispenseCandy = 1;
}
```

If/else Statements

If/else statements execute one of two statements depending on a condition, as shown below. When the expression in the if statement is TRUE, statement1 is executed. Otherwise, statement2 is executed.

```
if (expression)
   statement1
else
   statement2
```

C Code Example eC.6(b) gives an example if/else statement in C. The code sets y equal to a if a is greater than b; otherwise y = b.

Switch/case Statements

Switch/case statements execute one of several statements depending on the conditions, as shown in the general format below.

```
switch (variable) {
  case (expression1): statement1 break;
  case (expression2): statement2 break;
  case (expression3): statement3 break;
  default: statement4
}
```

For example, if `variable` is equal to `expression2`, execution continues at `statement2` until the keyword `break` is reached, at which point it exits the switch/case statement. If no conditions are met, the `default` executes.

If the keyword `break` is omitted, execution begins at the point where the condition is TRUE and then falls through to execute the remaining cases below it. This is usually not what you want and is a common error among beginning C programmers.

C Code Example eC.13 shows a switch/case statement that, depending on the variable `option`, determines the amount of money `amt` to be disbursed. A switch/case statement is equivalent to a series of nested if/else statements, as shown by the equivalent code in C Code Example eC.14.

C Code Example eC.13 SWITCH/CASE STATEMENT

```
// Assign amt depending on the value of option
switch (option) {
  case 1:   amt = 100; break;
  case 2:   amt = 50;  break;
  case 3:   amt = 20;  break;
  case 4:   amt = 10;  break;
  default: printf("Error: unknown option.\n");
}
```

C Code Example eC.14 NESTED IF/ELSE STATEMENT

```
// Assign amt depending on the value of option
if      (option == 1) amt = 100;
else if (option == 2) amt = 50;
else if (option == 3) amt = 20;
else if (option == 4) amt = 10;
else printf("Error: unknown option.\n");
```

C.7.2 Loops

While, do/while, and for loops are common loop constructs used in many high-level languages, including C. These loops repeatedly execute a statement as long as a condition is satisfied.

While Loops

While loops repeatedly execute a statement until a condition is not met, as shown in the general format below.

```
while (condition)
  statement
```

The while loop in C Code Example eC.15 computes the factorial of 9 = $9 \times 8 \times 7 \times \ldots \times 1$. Note that the condition is checked before executing the statement. In this example, the statement is a compound statement or block, so curly braces are required.

C Code Example eC.15 WHILE LOOP

```c
// Compute 9! (the factorial of 9)
int i = 1, fact = 1;

// Multiply the numbers from 1 to 9
while (i < 10) { // while loops check the condition first
  fact *= i;
  i++;
}
```

Do/while Loops

Do/while loops are like while loops but the condition is checked only after the statement is executed once. The general format is shown below. The condition is followed by a semi-colon.

```
do
  statement
while (condition);
```

The do/while loop in C Code Example eC.16 queries a user to guess a number. The program checks the condition (if the user's number is equal to the correct number) only after the body of the do/while loop executes once. This construct is useful when, as in this case, something must be done (e.g., the guess retrieved from the user) before the condition is checked. The scanf function, which is discussed in Section C.9.1, puts the value of the key pressed by a user into the variable guess.

C Code Example eC.16 DO/WHILE LOOP

```
// Query user to guess a number and check it against the correct number.
#define MAXGUESSES 3
#define CORRECTNUM 7

int guess, numGuesses = 0;
do {
  printf("Guess a number between 0 and 9. You have %d more guesses.\n",
         (MAXGUESSES-numGuesses));
  scanf("%d", &guess);      // Read user input
  numGuesses++;
} while ( (numGuesses < MAXGUESSES) & (guess != CORRECTNUM) );
// do loop checks the condition after the first iteration

if (guess == CORRECTNUM)
  printf("You guessed the correct number!\n");
```

For Loops

For loops, like while and do/while loops, repeatedly execute a statement until a condition is not satisfied. However, for loops add support for a *loop variable*, which typically keeps track of the number of loop executions. The general format of the for loop is

```
for (initialization; condition; loop operation)
  statement
```

The initialization code executes only once, before the for loop begins. The condition is tested at the beginning of each iteration of the loop. If the condition is not TRUE, the loop exits. The loop operation executes at the end of each iteration. C Code Example eC.17 shows the factorial of 9 computed using a for loop.

C Code Example eC.17 FOR LOOP

```
// Compute 9!
int i; // loop variable
int fact = 1;

for (i = 1; i < 10; i++)
  fact *= i;
```

Whereas the while and do/while loops in C Code Examples eC.15 and eC.16 include code for incrementing and checking the loop variable i and numGuesses, respectively, the for loop incorporates those statements into its format. A for loop could be expressed equivalently, but less conveniently, as

```
initialization;
while (condition) {
  statement
  loop operation;
}
```

SUMMARY

▸ **Control-flow statements:** C provides control-flow statements for conditional statements and loops.

▸ **Conditional statements:** Conditional statements execute a statement when a condition is TRUE. C includes the following conditional statements: if, if/else, and switch/case.

▸ **Loops:** Loops repeatedly execute a statement until a condition is FALSE. C provides while, do/while, and for loops.

C.8 MORE DATA TYPES

Beyond various sizes of integers and floating-point numbers, C includes other special data types, including pointers, arrays, strings, and structures. These data types are introduced in this section along with dynamic memory allocation.

C.8.1 Pointers

A pointer is the address of a variable. C Code Example eC.18 shows how to use pointers. salary1 and salary2 are variables that can contain integers, and ptr is a variable that can hold the address of an integer. The compiler will assign arbitrary locations in RAM for these variables, depending on the runtime environment. For the sake of concreteness, suppose this program is compiled on a 32-bit system with salary1 at addresses 0x70–73, salary2 at addresses 0x74–77, and ptr at 0x78–7B. Figure eC.3 shows memory and its contents after the program is executed.

In a variable declaration, a star (*) before a variable name indicates that the variable is a pointer to the declared type. In using a pointer variable, the * operator *dereferences* a pointer, returning the value stored at
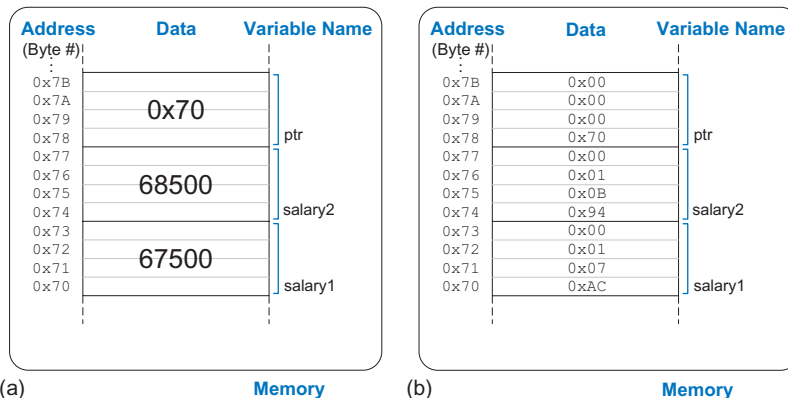
Figure eC.3 Contents of memory after C Code Example eC.18 executes shown (a) by value and (b) by byte using little-endian memory

the indicated memory address contained in the pointer. The & operator is pronounced "address of," and it produces the memory address of the variable being referenced.

C Code Example eC.18 POINTERS

```
// Example pointer manipulations
int salary1, salary2; // 32-bit numbers
int *ptr;              // a pointer specifying the address of an int variable

salary1 = 67500;       // salary1 = $67,500 = 0x000107AC
ptr = &salary1;        // ptr = 0x0070, the address of salary1
salary2 = *ptr + 1000; /* dereference ptr to give the contents of address 70 = $67,500,
                          then add $1,000 and set salary2 to $68,500 */
```

Pointers are particularly useful when a function needs to modify a variable instead of just returning a value. Because functions can't modify their inputs directly, a function can make the input a pointer to the variable. This is called passing an input variable *by reference* instead of *by value*, as shown in prior examples. C Code Example eC.19 gives an example of passing x by reference so that quadruple can modify the variable directly.

> Dereferencing a pointer to a nonexistent memory location or an address outside of the range accessible by the program will usually cause a program to crash. The crash is often called a *segmentation fault*.

C Code Example eC.19 PASSING AN INPUT VARIABLE BY REFERENCE

```
// Quadruple the value pointed to by a
#include <stdio.h>

void quadruple(int *a) {
  *a = *a * 4;
}

int main(void) {
  int x = 5;
  printf("x before: %d\n", x);
  quadruple(&x);
  printf("x after: %d\n", x);
  return 0;
}
```

Console Output

```
x before: 5
x after: 20
```

A pointer to address 0 is called a *null pointer* and indicates that the pointer is not actually pointing to meaningful data. It is written as NULL in a program.

C.8.2 Arrays

An array is a group of similar variables stored in consecutive addresses in memory. The elements are numbered from 0 to $N - 1$, where N is the length of the array. C Code Example eC.20 declares an array variable called scores that holds the final exam scores for three students. Memory space is reserved for three 32-bit integers, that is, $3 \times 4 = 12$ bytes. Suppose the scores array starts at address 0x40. The address of the 0th element (i.e., scores[0]) is 0x40, the first element is 0x44, and the second element is 0x48, as shown in Figure eC.4. In C, the array variable, in this case scores, is a pointer to the 0th element; in other words, scores contains the address of the 0th array element. It is the programmer's responsibility not to access elements beyond the end of the array. C has no internal bounds checking, so a program that writes beyond the end of an array will compile fine but may stomp on other parts of memory when it runs.

Colloquially, the "0th" array element is also referred to as the "first" array element.

C Code Example eC.20 ARRAY DECLARATION

```
int32_t scores[3];    // array of three 4-byte numbers
```

The elements of an array can be initialized either at declaration using curly braces {}, as shown in C Code Example eC.21, or individually in the body of the code, as shown in C Code Example eC.22. Each element of an array is accessed using brackets []. The contents of memory containing the array are shown in Figure eC.4. Array initialization using curly braces {} can be performed only at declaration and not afterward. for loops are commonly used to assign and read array data, as shown in C Code Example eC.23.

C Code Example eC.21 ARRAY INITIALIZATION AT DECLARATION USING {}

```
int32_t scores[3] = {93, 81, 97}; // scores[0] = 93; scores[1] = 81; scores[2] = 97;
```
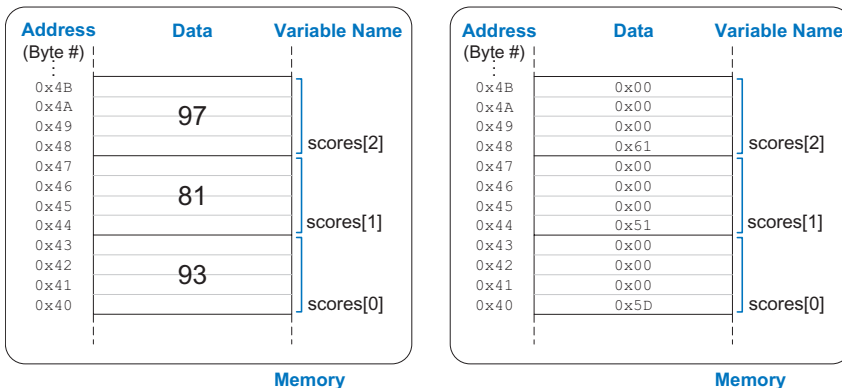

Figure eC.4 scores **array stored in memory**

C Code Example eC.22 ARRAY INITIALIZATION USING ASSIGNMENT

```
int32_t scores[3];

scores[0] = 93;
scores[1] = 81;
scores[2] = 97;
```

C Code Example eC.23 ARRAY INITIALIZATION USING A `for` LOOP

```
// User enters 3 student scores into an array
int32_t scores[3];
int i, entered;

printf("Please enter the student's 3 scores.\n");
for (i = 0; i < 3; i++) {
  printf("Enter a score and press enter.\n");
  scanf("%d", &entered);
  scores[i] = entered;
}
printf("Scores: %d %d %d\n", scores[0], scores[1], scores[2]);
```

When an array is declared, the length must be constant so that the compiler can allocate the proper amount of memory. However, when the array is passed to a function as an input argument, the length need not be defined because the function only needs to know the address of the beginning of the array. C Code Example eC.24 shows how an array is passed to a function. The input argument `arr` is simply the address of the 0th element of an array. Often, the number of elements in an array is also passed as an input argument. In a function, an input argument of type `int[]` indicates that it is an array of integers. Arrays of any type may be passed to a function.

C Code Example eC.24 PASSING AN ARRAY AS AN INPUT ARGUMENT

```
// Initialize a 5-element array, compute the mean, and print the result.
#include <stdio.h>

// Returns the mean value of an array (arr) of length len
float getMean(int arr[], int len) {
  int i;
  float mean, total = 0;

  for (i=0; i < len; i++)
    total += arr[i];

  mean = total / len;
  return mean;
}
int main(void) {
  int data[4] = {78, 14, 99, 27};
```

```
  float avg;
  avg = getMean(data, 4);
  printf("The average value is: %f.\n", avg);
}
```

Console Output

```
The average value is: 54.500000.
```

An array argument is equivalent to a pointer to the beginning of the array. Thus, getMean could also have been declared as

```
float getMean(int *arr, int len);
```

Although functionally equivalent, datatype[] is the preferred method for passing arrays as input arguments because it more clearly indicates that the argument is an array.

A function is limited to a single output, that is, return variable. However, by receiving an array as an input argument, a function can essentially output more than a single value by changing the array itself. C Code Example eC.25 sorts an array from lowest to highest and leaves the result in the same array. The three function prototypes below are equivalent. The length of an array in a function declaration (i.e., int vals[100]) is ignored.

```
void sort(int *vals, int len);
void sort(int vals[], int len);
void sort(int vals[100], int len);
```

C Code Example eC.25 PASSING AN ARRAY AND ITS LENGTH AS INPUTS

```c
// Sort the elements of the array vals of length len from lowest to highest
void sort(int vals[], int len) {
  int i, j, temp;
  for (i = 0; i < len; i++) {
    for (j = i + 1; j < len; j++) {
      if (vals[i] > vals[j]) {
        temp = vals[i];
        vals[i] = vals[j];
        vals[j] = temp;
      }
    }
  }
}
```

Arrays may have multiple dimensions. C Code Example eC.26 uses a two-dimensional array to store the grades across eight problem sets for ten students. Recall that initialization of array values using { } is allowed only at declaration.

C Code Example eC.26 TWO-DIMENSIONAL ARRAY INITIALIZATION

```
// Initialize 2D array at declaration
 int grades[10][8] = { {100,  107, 99,   101, 100, 104, 109, 117},
                       {103, 101, 94,   101, 102, 106, 105, 110},
                       {101, 102, 92,   101, 100, 107, 109, 110},
                       {114, 106, 95,   101, 100, 102, 102, 100},
                       {98,   105, 97,   101, 103, 104, 109, 109},
                       {105, 103, 99,   101, 105, 104, 101, 105},
                       {103, 101, 100, 101, 108, 105, 109, 100},
                       {100, 102, 102, 101, 102, 101, 105, 102},
                       {102, 106, 110, 101, 100, 102, 120, 103},
                       {99,   107, 98,   101, 109, 104, 110, 108} };
```

C Code Example eC.27 shows some functions that operate on the 2D grades array from C Code Example eC.26. Multidimensional arrays used as input arguments to a function must define all but the first dimension. Thus, the following two function prototypes are acceptable:

```
void print2dArray(int arr[10][8]);
void print2dArray(int arr[][8]);
```

C Code Example eC.27 OPERATING ON MULTIDIMENSIONAL ARRAYS

```c
#include <stdio.h>

// Print the contents of a 10 × 8 array
void print2dArray(int arr[10][8]) {
 int i, j;
 for (i = 0; i < 10; i++) {        // for each of the 10 students
   printf("Row %d\n", i);
   for (j = 0; j < 8; j++) {
     printf("%d ", arr[i][j]); // print scores for all 8 problem sets
   }
   printf("\n");
 }
}

// Calculate the mean score of a 10 × 8 array
float getMean(int arr[10][8]) {
 int i, j;
 float mean, total = 0;
 // get the mean value across a 2D array
 for (i = 0; i < 10; i++) {
   for (j = 0; j < 8; j++) {
     total += arr[i][j];       // sum array values
   }
 }
 mean = total / (10 * 8);
 printf("Mean is: %f\n", mean);
 return mean;
}
```

Note that because an array is represented by a pointer to the initial element, C cannot copy or compare arrays using the = or == operators. Instead, you must use a loop to copy or compare each element one at a time.

C.8.3 Characters

A character (char) is an 8-bit variable. It can be viewed either as a two's complement number between −128 and 127 or as an ASCII code for a letter, digit, or symbol. ASCII characters can be specified as a numeric value (in decimal, hexadecimal, etc.) or as a printable character enclosed in single quotes. For example, the letter A has the ASCII code 0x41, B = 0x42, etc. Thus "A" + 3 is 0x44, or "D." Table 6.2 lists the ASCII character encodings, and Table eC.4 lists characters used to indicate formatting or special characters. Formatting codes include carriage return (\r), newline (\n), horizontal tab (\t), and the end of a string (\0). \r is shown for completeness but is rarely used in C programs. \r returns the carriage (location of typing) to the beginning (left) of the line, but any text that was there is overwritten. \n, instead, moves the location of typing to the beginning of a new line.[3] The *NULL* character ('\0') indicates the end of a text string and is discussed next in Section C.8.4.

C.8.4 Strings

A string is an array of characters used to store a piece of text of bounded but variable length. Each character is a byte representing the ASCII code

Table eC.4 Special characters

Special Character	Hexadecimal Encoding	Description
\r	0x0D	carriage return
\n	0x0A	new line
\t	0x09	tab
\0	0x00	terminates a string
\\	0x5C	backslash
\"	0x22	double quote
\'	0x27	single quote
\a	0x07	bell

The term "carriage return" originates from typewriters that required the carriage, the contraption that holds the paper, to move to the right in order to allow typing to begin at the left side of the page. A carriage return lever, shown on the left in the figure below, is pressed so that the carriage would both move to the right and advance the paper by one line, called a line feed.

A Remington electric typewriter used by Winston Churchill. http://cwr.iwm.org.uk/server/show/conMediaFile.71979

[3] Windows text files use \r\n to represent end-of-line while UNIX-based systems use \n, which can cause nasty bugs when moving text files between systems.

C strings are called *null terminated* or *zero terminated* because the length is determined by looking for a zero at the end. In contrast, languages such as Pascal use the first byte to specify the string length, up to a maximum of 255 characters. This byte is called the *prefix byte* and such strings are called *P-strings*. An advantage of null-terminated strings is that the length can be arbitrarily great. An advantage of P-strings is that the length can be determined immediately without having to inspect all of the characters of the string.

for that letter, number, or symbol. The size of the array determines the maximum length of the string, but the actual length of the string could be shorter. In C, the length of the string is determined by looking for the NULL terminator (ASCII value 0x00) at the end of the string.

C Code Example eC.28 shows the declaration of a 10-element character array called `greeting` that holds the string `"Hello!"` For concreteness, suppose `greeting` starts at memory address 0x50. Figure eC.5 shows the contents of memory from 0x50 to 0x59 holding the string `"Hello!"` Note that the string uses only the first seven elements of the array even though ten elements are allocated in memory.

C Code Example eC.28 STRING DECLARATION

```
char greeting[10] = "Hello!";
```

C Code Example eC.29 shows an alternate declaration of the string `greeting`. The pointer `greeting` holds the address of the first element of a 7-element array composed of each of the characters in `"Hello!"` followed by the null terminator. The code also demonstrates how to print strings by using the `%s` (string) format code.

C Code Example eC.29 ALTERNATE STRING DECLARATION

```
char *greeting = "Hello!";
printf("greeting: %s", greeting);
```

Console Output

```
greeting: Hello!
```

Unlike primitive variables, a string cannot be set equal to another string using the equals operator, =. Each element of the character array must be individually copied from the source string to the target string. This is true for any array. C Code Example eC.30 copies one string, `src`, to another, `dst`. The sizes of the arrays are not needed because the end of the `src` string is indicated by the null terminator. However, `dst` must be large enough so that you don't stomp on other data. `strcpy` and other string manipulation functions are available in C's built-in libraries (see Section C.9.4).

C Code Example eC.30 COPYING STRINGS

```
// Copy the source string, src, to the destination string, dst
void strcpy(char *dst, char *src) {
  int i = 0;

  do {
    dst[i] = src[i];        // copy characters one byte at a time
  } while (src[i++]);       // until the null terminator is found
{
```
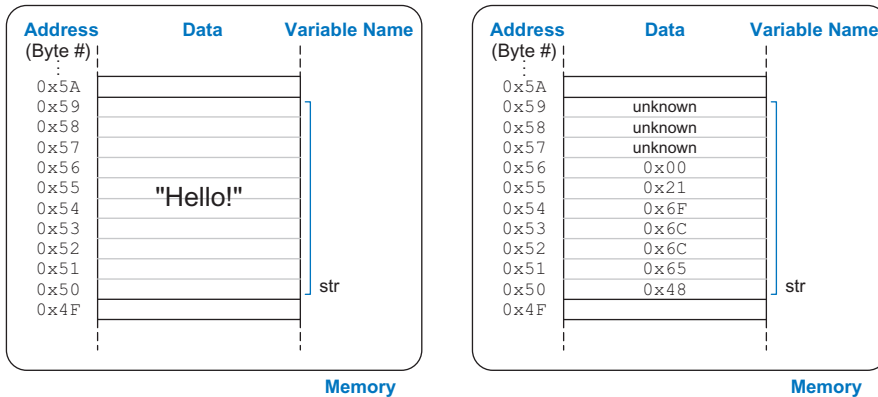
Figure eC.5 The string "Hello!" stored in memory

C.8.5 Structures

In C, structures are used to store a collection of data of various types. The general format of a structure declaration is

```
struct name {
  type1 element1;
  type2 element2;
  ...
};
```

struct is a keyword indicating that it is a structure; name is the structure tag name; and element1 and element2 are members of the structure. A structure may have any number of members. C Code Example eC.31 shows how to use a structure to store contact information. The program then declares a variable c1 of type struct contact.

C Code Example eC.31 STRUCTURE DECLARATION

```
struct contact {
  char name[30];
  int phone;
  float height; // in meters
};

struct contact c1;
strcpy(c1.name, "Ben Bitdiddle");
c1.phone = 7226993;
c1.height = 1.82;
```

Just like built-in C types, you can create arrays of structures and pointers to structures. C Code Example eC.32 creates an array of contacts.

C Code Example eC.32 ARRAY OF STRUCTURES

```
struct contact classlist[200];
classlist[0].phone = 9642025;
```

It is common to use pointers to structures. C provides the *member access operator* `->` to dereference a pointer to a structure and access a member of the structure. C Code Example eC.33 shows an example of declaring a pointer to a `struct contact`, assigning it to point to the 42nd element of `classlist` from C Code Example eC.32, and using the member access operator to set a value in that element.

C Code Example eC.33 ACCESSING STRUCTURE MEMBERS USING POINTERS AND `->`

```
struct contact *cptr;
cptr = &classlist[42];
cptr -> height = 1.9; // equivalent to: (*cptr).height = 1.9;
```

Structures can be passed as function inputs or outputs by value or by reference. Passing by value requires the compiler to copy the entire structure into memory for the function to access. This can require a large amount of memory and time for a big structure. Passing by reference involves passing a pointer to the structure, which is more efficient. The function can also modify the structure being pointed to rather than having to return another structure. C Code Example eC.34 shows two versions of the `stretch` function that makes a contact 2 cm taller. `stretchByReference` avoids copying the large structure twice.

C Code Example eC.34 PASSING STRUCTURES BY VALUE OR BY NAME

```
struct contact stretchByValue(struct contact c) {
  c.height += 0.02;
  return c;
}
void stretchByReference(struct contact *cptr) {
  cptr -> height += 0.02;
}
int main(void) {
  struct contact George;

  George.height = 1.4; // poor fellow has been stooped over
  George = stretchByValue(George); // stretch for the stars
  stretchByReference(&George);     // and stretch some more
}
```

C.8.6 `typedef`

C also allows you to define your own names for data types using the `typedef` statement. For example, writing `struct contact` becomes tedious when it is used often, so we can define a new type named `contact` and use it as shown in C Code Example eC.35.

C Code Example eC.35 **CREATING A CUSTOM TYPE USING** `typedef`

```
typedef struct contact {
  char name[30];
  int phone;
  float height;  // in meters
} contact;         // defines contact as shorthand for "struct contact"

contact c1;        // now we can declare the variable as type contact
```

`typedef` can be used to create a new type occupying the same amount of memory as a primitive type. C Code Example eC.36 defines `byte` and `bool` as 8-bit types. The `byte` type may make it clearer that the purpose of `pos` is to be an 8-bit number rather than an ASCII character. The `bool` type indicates that the 8-bit number is representing TRUE or FALSE. These types make a program easier to read than if one simply used `char` everywhere.

C Code Example eC.36 `typedef` `byte` **AND** `bool`

```
typedef unsigned char byte;
typedef char bool;
#define TRUE 1
#define FALSE 0

byte pos = 0x45;
bool loveC = TRUE;
```

C Code Example eC.37 illustrates defining a 3-element vector and a 3×3 matrix type using arrays.

C Code Example eC.37 `typedef` `vector` **AND** `matrix`

```
typedef double vector[3];
typedef double matrix[3][3];

vector a = {4.5, 2.3, 7.0};
matrix b = {{3.3, 4.7, 9.2}, {2.5, 4, 9}, {3.1, 99.2, 88}};
```

C.8.7 Dynamic Memory Allocation*

In all examples thus far, variables have been declared *statically*; that is, their size is known at compile time. This can be problematic for arrays and strings of variable size because the array must be declared large enough to accommodate the largest size the program will ever see. An alternative is to *dynamically* allocate memory at run time when the actual size is known.

The malloc function from stdlib.h allocates a block of memory of a specified size and returns a pointer to it. If not enough memory is available, it returns a NULL pointer instead. For example, the following code allocates ten 16-bit integers ($10 \times 2 = 20$ bytes). The sizeof operator returns the size of a type or variable in bytes.

```
// Dynamically allocate 20 bytes of memory
int16_t *data = malloc(10 * sizeof(int16_t));
```

C Code Example eC.38 illustrates dynamic allocation and deallocation. The program accepts a variable number of inputs, stores them in a dynamically allocated array, and computes their average. The amount of memory necessary depends on the number of elements in the array and the size of each element. For example, if an int is a 4-byte variable and ten elements are needed, 40 bytes are dynamically allocated. The free function deallocates the memory so that it could later be used for other purposes. Failing to deallocate dynamically allocated memory is called a *memory leak* and should be avoided.

C Code Example eC.38 DYNAMIC MEMORY ALLOCATION
AND DEALLOCATION

```
// Dynamically allocate and deallocate an array using malloc and free
#include <stdlib.h>
// Insert getMean function from C Code Example eC.24.
int main(void) {
  int len, i;
  int *nums;
  printf("How many numbers would you like to enter? ");
  scanf("%d", &len);
  nums = malloc(len*sizeof(int));
  if (nums == NULL) printf("ERROR: out of memory.\n");
  else {
    for (i = 0; i < 100; i++) {
      printf("Enter number: ");
      scanf("%d", &nums[i]);
    }
    printf("The average is %f\n", getMean(nums, len));
  }
  free(nums);
}
```

C.8.8 Linked Lists*

A *linked list* is a common data structure used to store a variable number of elements. Each element in the list is a structure containing one or more data fields and a link to the next element. The first element in the list is called the *head*. Linked lists illustrate many of the concepts of structures, pointers, and dynamic memory allocation.

C Code Example eC.39 describes a linked list for storing computer user accounts to accommodate a variable number of users. Each user has a user name, password, unique user identification number (UID), and a field indicating whether they have administrator privileges. Each element of the list is of type userL, containing all of this user information along with a link to the next element in the list. A pointer to the head of the list is stored in a global variable called users, and is initially set to NULL to indicate that there are no users.

The program defines functions to insert, delete, and find a user and to count the number of users. The insertUser function allocates space for a new list element and adds it to the head of the list. The deleteUser function scans through the list until the specified UID is found and then removes that element, adjusting the link from the previous element to skip the deleted element and freeing the memory occupied by the deleted element. The findUser function scans through the list until the specified UID is found and returns a pointer to that element, or NULL if the UID is not found. The numUsers function counts the number of elements in the list.

C Code Example eC.39 LINKED LIST

```c
#include <stdlib.h>
#include <string.h>

typedef struct userL {
  char uname[80];      // user name
  char passwd[80];     // password
  int uid;             // user identification number
  int admin;           // 1 indicates administrator privileges
  struct userL *next;
} userL;

userL *users = NULL;

void insertUser(char *uname, char *passwd, int uid, int admin) {
  userL *newUser;

  newUser = malloc(sizeof(userL));   // create space for new user
  strcpy(newUser->uname, uname);     // copy values into user fields
  strcpy(newUser->passwd, passwd);
  newUser->uid = uid;
  newUser->admin = admin;
  newUser->next = users;             // insert at start of linked list
  users = newUser;
}
```

```
void deleteUser(int uid) {   // delete first user with given uid
  userL *cur = users;
  userL *prev = NULL;

  while (cur != NULL) {
    if (cur->uid == uid) { // found the user to delete
      if (prev == NULL) users = cur->next;
      else prev->next = cur->next;
      free(cur);
      return; // done
    }
    prev = cur;      // otherwise, keep scanning through list
    cur = cur->next;
  }
}

userL *findUser(int uid) {
  userL *cur = users;

  while (cur != NULL) {
    if (cur->uid == uid) return cur;
    else cur = cur->next;
  }
  return NULL;
}

int numUsers(void) {
  userL *cur = users;
  int count = 0;

  while (cur != NULL) {
    count++;
    cur = cur->next;
  }
  return count;
}
```

SUMMARY

- ▶ **Pointers:** A pointer holds the address of a variable.

- ▶ **Arrays:** An array is a list of similar elements declared using square brackets [].

- ▶ **Characters:** char types can hold small integers or special codes for representing text or symbols.

- ▶ **Strings:** A string is an array of characters ending with the null terminator 0x00.

- ▶ **Structures:** A structure stores a collection of related variables.

- ▶ **Dynamic memory allocation:** malloc is a built-in function in the standard library (stdlib.h) for allocating memory as the program runs. free deallocates the memory after use.

- ▶ **Linked Lists:** A linked list is a common data structure for storing a variable number of elements.

C.9 STANDARD LIBRARIES

Programmers commonly use a variety of standard functions, such as printing and trigonometric operations. To save each programmer from having to write these functions from scratch, C provides *libraries* of frequently used functions. Each library has a header file and an associated object file, which is a partially compiled C file. The header file holds variable declarations, defined types, and function prototypes. The object file contains the functions themselves and is linked at compile time to create the executable. Because the library function calls are already compiled into an object file, compile time is reduced. Table eC.5 lists some of the most frequently used C libraries, and each is described briefly below.

C.9.1 stdio

The standard input/output library `stdio.h` contains commands for printing to a console, reading keyboard input, and reading and writing files. To use these functions, the library must be included at the top of the C file:

```
#include <stdio.h>
```

Table eC.5 Frequently used C libraries

C Library Header File	Description
stdio.h	**Standard input/output library**. Includes functions for printing or reading to/from the screen or a file (`printf`, `fprintf` and `scanf`, `fscanf`) and to open and close files (`fopen` and `fclose`).
stdlib.h	**Standard library**. Includes functions for random number generation (`rand` and `srand`), for dynamically allocating or freeing memory (`malloc` and `free`), terminating the program early (`exit`), and for conversion between strings and numbers (`atoi`, `atol`, and `atof`).
math.h	**Math library**. Includes standard math functions such as `sin`, `cos`, `asin`, `acos`, `sqrt`, `log`, `log10`, `exp`, `floor`, and `ceil`.
string.h	**String library**. Includes functions to compare, copy, concatenate, and determine the length of strings.

printf

The *print formatted* function `printf` displays text to the console. Its required input argument is a string enclosed in quotes `""`. The string contains text and optional commands to print variables. Variables to be printed are listed after the string and are printed using format codes shown in Table eC.6. C Code Example eC.40 gives a simple example of `printf`.

Table eC.6 printf format codes for printing variables

Code	Format
%d	Decimal
%u	Unsigned decimal
%x	Hexadecimal
%o	Octal
%f	Floating-point number (`float` or `double`)
%e	Floating-point number (`float` or `double`) in scientific notation (e.g., 1.56e7)
%c	Character (`char`)
%s	String (null-terminated array of characters)

C Code Example eC.40 PRINTING TO THE CONSOLE USING `printf`

```c
// Simple print function
#include <stdio.h>

int num = 42;
int main(void) {
   printf("The answer is %d.\n", num);
}
```

Console Output:

```
The answer is 42.
```

Floating-point formats (`floats` and `doubles`) default to printing six digits after the decimal point. To change the precision, replace `%f` with `%w.df`, where `w` is the minimum width of the number and `d` is the number of decimal places to print. Note that the decimal point is included in the width count. In C Code Example eC.41, `pi` is printed with a total

of four characters, two of which are after the decimal point: 3.14. e is printed with a total of eight characters, three of which are after the decimal point. Because it has only one digit before the decimal point, it is padded with three leading spaces to reach the requested width. c should be printed with five characters, three of which are after the decimal point. But it is too wide to fit, so the requested width is overridden while retaining the three digits after the decimal point.

C Code Example eC.41 FLOATING POINT NUMBER FORMATS FOR PRINTING

```
// Print floating-point numbers with different formats
float pi = 3.14159, e = 2.7182, c = 2.998e8;
printf("pi = %4.2f\ne = %8.3f\nc = %5.3f\n", pi, e, c);
```

Console Output:

```
pi = 3.14
e  = 2.718
c  = 299800000.000
```

Because % and \ are used in print formatting, to print these characters themselves, you must use the special character sequences shown in C Code Example eC.42.

C Code Example eC.42 PRINTING % AND \ USING `printf`

```
// How to print % and \ to the console
printf("Here are some special characters: %% \\ \n");
```

Console Output:

```
Here are some special characters: % \
```

scanf

The scanf function reads text typed on the keyboard. It uses format codes in the same way as `printf`. C Code Example eC.43 shows how to use `scanf`. When the `scanf` function is encountered, the program waits until the user types a value before continuing execution. The arguments to `scanf` are a string indicating one or more format codes and pointers to the variables where the results should be stored.

C Code Example eC.43 READING USER INPUT FROM THE KEYBOARD
WITH `scanf`

```
// Read variables from the command line
#include <stdio.h>

int main(void)
{
int a;
   char str[80];
   float f;

   printf("Enter an integer.\n");
   scanf("%d", &a);
   printf("Enter a floating-point number.\n");
   scanf("%f", &f);
   printf("Enter a string.\n");
   scanf("%s", str);   // note no & needed: str is a pointer
}
```

File Manipulation

Many programs need to read and write files, either to manipulate data already stored in a file or to log large amounts of information. In C, the file must first be opened with the `fopen` function. It can then be read or written with `fscanf` or `fprintf` in a way analogous to reading and writing to the console. Finally, it should be closed with the `fclose` command.

The `fopen` function takes as arguments the file name and a *print mode*. It returns a *file pointer* of type `FILE*`. If `fopen` is unable to open the file, it returns NULL. This might happen when one tries to read a nonexistent file or write a file that is already opened by another program. The modes are:

`"w"`: Write to a file. If the file exists, it is overwritten.

`"r"`: Read from a file.

`"a"`: Append to the end of an existing file. If the file doesn't exist, it is created.

C Code Example eC.44 shows how to open, print to, and close a file. It is good practice to always check whether the file was opened successfully and to provide an error message if it was not. The `exit` function will be discussed in Section C.9.2. The `fprintf` function is like `printf` but it also takes the file pointer as an input argument to know which file to write. `fclose` closes the file, ensuring that all of the information is actually written to disk and freeing up file system resources.

C Code Example eC.44 PRINTING TO A FILE USING `fprintf`

```
// Write "Testing file write." to result.txt
#include <stdio.h>
#include <stdlib.h>
int main(void) {
  FILE *fptr;
  if((fptr = fopen("result.txt", "w")) == NULL) {
    printf("Unable to open result.txt for writing.\n");
    exit(1); // exit the program indicating unsuccessful execution
  }
  fprintf(fptr, "Testing file write.\n");
  fclose(fptr);
}
```

It is idiomatic to open a file and check whether the file pointer is NULL in a single line of code, as shown in C Code Example eC.44. However, you could just as easily separate the functionality into two lines:

```
fptr=fopen("result.txt","w");
if (fptr == NULL)
 ...
```

C Code Example eC.45 illustrates reading numbers from a file named data.txt using `fscanf`. The file must first be opened for reading. The program then uses the `feof` function to check whether it has reached the end of the file. As long as the program is not at the end, it reads the next number and prints it to the screen. Again, the program closes the file at the end to free up resources.

C Code Example eC.45 READING INPUT FROM A FILE USING `fscanf`

```
#include <stdio.h>
int main(void) {
  FILE *fptr;
  int data;
  // read in data from input file
  if ((fptr = fopen("data.txt", "r")) == NULL) {
    printf("Unable to read data.txt\n");
    exit(1);
  }
  while (!feof(fptr)) { // check that the end of the file hasn't been reached
    fscanf(fptr, "%d", &data);
    printf("Read data: %d\n", data);
  }
  fclose(fptr);
}
```

data.txt

```
25 32 14 89
```

Console Output:

```
Read data: 25
Read data: 32
Read data: 14
Read data: 89
```

Other Handy stdio Functions

The sprintf function prints characters into a string, and sscanf reads variables from a string. The fgetc function reads a single character from a file, while fgets reads a complete line into a string.

fscanf is rather limited in its ability to read and parse complex files, so it is often easier to fgets one line at a time and then digest that line using sscanf or with a loop that inspects characters one at a time using fgetc.

C.9.2 stdlib

The standard library stdlib.h provides general-purpose functions, including random-number generation (rand and srand), dynamic memory allocation (malloc and free, already discussed in Section C.8.7), exiting the program early (exit), and number format conversions. To use these functions, add the following line at the top of the C file.

```
#include <stdlib.h>
```

rand and srand

rand returns a pseudo-random integer. Pseudo-random numbers have the statistics of random numbers but follow a deterministic pattern starting with an initial value called the *seed*. To convert the number to a particular range, use the modulo operator (%) as shown in C Code Example eC.46 for a range of 0 to 9. The values x and y will be random but they will be the same each time this program runs. Sample console output is given below the code.

C Code Example eC.46 RANDOM NUMBER GENERATION USING rand

```
#include <stdlib.h>
int x, y;
x = rand();      // x = a random integer
y = rand() % 10;  // y = a random number from 0 to 9
printf("x = %d, y = %d\n", x, y);
```

Console Output:

```
x = 1481765933, y = 3
```

A programmer creates a different sequence of random numbers each time a program runs by changing the seed. This is done by calling the srand function, which takes the seed as its input argument. As shown in

C Code Example eC.47, the seed itself must be random, so a typical C program assigns it by calling the time function, which returns the current time in seconds.

C Code Example eC.47 SEEDING THE RANDOM NUMBER GENERATOR USING srand

```
// Produce a different random number each run
#include <stdlib.h>
#include <time.h>        // needed to call time()

int main(void) {
  int x;

  srand(time(NULL));    // seed the random number generator
  x = rand() % 10;      // random number from 0 to 9
  printf("x = %d\n", x);
}
```

exit

The exit function terminates a program early. It takes a single argument that is returned to the operating system to indicate the reason for termination. 0 indicates normal completion, while nonzero conveys an error condition.

Format Conversion: atoi, atol, atof

The standard library provides functions for converting ASCII strings to integers or doubles using atoi and atof, as shown in C Code Example eC.48. This is particularly useful when reading in mixed data (a mix of strings and numbers) from a file or when processing numeric command line arguments, as described in Section C.10.3.

C Code Example eC.48 FORMAT CONVERSION

```
// Convert ASCII strings to ints and floats
#include <stdlib.h>

int main(void) {
  int x;
  double z;

  x = atoi("42");
  z = atof("3.822");

  printf("x = %d\tz = %f\n", x, z);
}
```

Console Output:

```
x = 42      z = 3.822000
```

For historical reasons, the time function usually returns the current time in seconds relative to January 1, 1970 00:00 UTC. UTC stands for Coordinated Universal Time, which is the same as Greenwich Mean Time (GMT). This date is just after the UNIX operating system was created by a group at Bell Labs, including Dennis Ritchie and Brian Kernighan, in 1969. Similar to New Year's Eve parties, some UNIX enthusiasts hold parties to celebrate significant values returned by time. For example, on February 1, 2009 at 23:31:30 UTC, time returned 1,234,567,890. In the year 2038, 32-bit UNIX clocks will overflow into the year 1901.

C.9.3 math

The math library `math.h` provides commonly used math functions such as trigonometry functions, square root, and logs. C Code Example eC.49 shows how to use some of these functions. To use math functions, place the following line in the C file:

```
#include <math.h>
```

C Code Example eC.49 MATH FUNCTIONS

```
// Example math functions
#include <stdio.h>
#include <math.h>

int main(void) {
  float a, b, c, d, e, f, g, h;

  a = cos(0);            // a = 1, note: the input argument is in radians
  b = 2 * acos(0);       // b = pi (acos means arc cosine)
  c = sqrt(144);         // c = 12
  d = exp(2);            // d = e^2 = 7.389056
  e = log(7.389056);     // e = 2 (natural logarithm, base e)
  f = log10(1000);       // f = 3 (log base 10)
  g = floor(178.567);    // g = 178, rounds to next lowest whole number
  h = pow(2, 10);        // h = 2^10 (i.e., 2 raised to the 10th power)

  printf("a = %.0f, b = %f, c = %.0f, d = %.0f, e = %.2f, f = %.0f, g = %.2f, h = %.2f\n",
         a, b, c, d, e, f, g, h);
}
```

Console Output:

```
a = 1, b = 3.141593, c = 12, d = 7, e = 2.00, f = 3, g = 178.00, h = 1024.00
```

C.9.4 string

The string library `string.h` provides commonly used string manipulation functions. Key functions include:

```
// Copy src into dst and return dst
char *strcpy(char *dst, char *src);

// Concatenate (append) src to the end of dst and return dst
char *strcat(char *dst, char *src);

// Compare two strings. Return 0 if equal, nonzero otherwise
int strcmp(char *s1, char *s2);

// Return the length of str, not including the null termination
int strlen(char *str);
```

C.10 COMPILER AND COMMAND LINE OPTIONS

Although we have introduced relatively simple C programs, real-world programs can consist of tens or even thousands of C files to enable modularity, readability, and multiple programmers. This section describes how to compile a program spread across multiple C files and shows how to use compiler options and command line arguments.

C.10.1 Compiling Multiple C Source Files

Multiple C files are compiled into a single executable by listing all file names on the compile line as shown below. Remember that the group of C files still must contain only one `main` function, conventionally placed in a file named `main.c`.

```
gcc main.c file2.c file3.c
```

C.10.2 Compiler Options

Compiler options allow the programmer to specify such things as output file names and formats, optimizations, etc. Compiler options are not standardized, but Table eC.7 lists ones that are commonly used. Each option is typically preceded by a dash (-) on the command line, as shown. For example, the "-o" option allows the programmer to specify an output file name other than the `a.out` default. A plethora of options exist; they can be viewed by typing `gcc --help` at the command line.

Table eC.7 Compiler options

Compiler Option	Description	Example
-o outfile	Specifies output file name	gcc -o hello hello.c
-S	Create assembly language output file (not executable)	gcc -S hello.c this produces hello.s
-v	Verbose mode—prints the compiler results and processes as compilation completes	gcc -v hello.c
-Olevel	Specify the optimization level (level is typically 0 through 3), producing faster and/or smaller code at the expense of longer compile time	gcc -O3 hello.c
--version	List the version of the compiler	gcc -version
--help	List all command line options	gcc --help
-Wall	Print all warnings	gcc -Wall hello.c

C.10.3 Command Line Arguments

Like other functions, `main` can also take input variables. However, unlike other functions, these arguments are specified at the command line. As shown in C Code Example eC.50, `argc` stands for *argument count*, and it denotes the number of arguments on the command line. `argv` stands for *argument vector*, and it is an array of the strings found on the command line. For example, suppose the program in C Code Example eC.50 is compiled into an executable called `testargs`. When the lines below are typed at the command line, `argc` has the value 4, and the array `argv` has the values {`"./testargs"`, `"arg1"`, `"25"`, `"lastarg!"`}. Note that the executable name is counted as the 0th (i.e., left-most) argument. The console output after typing this command is shown below C Code Example eC.50.

```
gcc -o testargs testargs.c
./testargs arg1 25 lastarg!
```

Programs that need numeric arguments may convert the string arguments to numbers using the functions in `stdlib.h`.

C Code Example eC.50 COMMAND LINE ARGUMENTS

```c
// Print command line arguments
#include <stdio.h>
int main(int argc, char *argv[]) {
  int i;
  for (i = 0; i < argc; i++)
    printf("argv[%d] = %s\n", i, argv[i]);
}
```

Console Output

```
argv[0] = ./testargs
argv[1] = arg1
argv[2] = 25
argv[3] = lastarg!
```

C.11 COMMON MISTAKES

As with any programming language, you are almost certain to make errors while you write nontrivial C programs. Below are descriptions of some common mistakes made when programming in C. Some of these errors are particularly troubling because they compile but do not function as the programmer intended.

C Code Mistake eC.1 MISSING & IN scanf

Erroneous Code	Corrected Code
```int a;```	```int a;```

```
int a;
printf("Enter an integer:\t");
scanf("%d", a); // missing & before a
```

```
int a;
printf("Enter an integer:\t");
scanf("%d", &a);
```

C Code Mistake eC.2 USING = INSTEAD OF == FOR COMPARISON

Erroneous Code

```
if (x = 1) // always evaluates as TRUE
  printf("Found!\n");
```

Corrected Code

```
if (x == 1)
   printf("Found!\n");
```

C Code Mistake eC.3 INDEXING PAST LAST ELEMENT OF ARRAY

Erroneous Code

```
int array[10];
array[10] = 42;      // index is 0-9
```

Corrected Code

```
int array[10];
array[9] = 42;
```

C Code Mistake eC.4 USING = IN #define STATEMENT

Erroneous Code

```
// replaces NUM with "= 4" in code
#define NUM = 4
```

Corrected Code

```
#define NUM 4
```

C Code Mistake eC.5 USING AN UNINITIALIZED VARIABLE

Erroneous Code

```
int i;
if (i == 10) // i is uninitialized
  ...
```

Corrected Code

```
int i = 10;
if (i == 10)
   ...
```

C Code Mistake eC.6 NOT INCLUDING PATH OF USER-CREATED HEADER FILES

Erroneous Code

```
#include "myfile.h"
```

Corrected Code

```
#include "othercode\myfile.h"
```

Debugging skills are acquired with practice, but here are a few hints.

- Fix bugs starting with the first error indicated by the compiler. Later errors may be downstream effects of this error. After fixing that bug, recompile and repeat until all bugs (at least those caught by the compiler!) are fixed.
- When the compiler says a valid line of code is in error, check the code above it (i.e., for missing semicolons or braces).
- When needed, split up complicated statements into multiple lines.
- Use printf to output intermediate results.
- When a result doesn't match expectations, start debugging the code at the *first* place it deviates from expectations.
- Look at all compiler warnings. While some warnings can be ignored, others may alert you to more subtle code errors that will compile but not run as intended.

C Code Mistake eC.7 USING LOGICAL OPERATORS (!, ||, &&) INSTEAD
OF BITWISE (~, |, &)

Erroneous Code	Corrected Code

```
char x = !5;     // logical NOT:  x = 0
char y = 5||2;   // logical OR:   y = 1
char z = 5&&2;   // logical AND:  z = 1
```

```
char x = ~5;  // bitwise NOT:  x = 0b11111010
char y = 5|2; // bitwise OR:   y = 0b00000111
char z = 5&2; // logical AND:  z = 0b00000000
```

C Code Mistake eC.8 FORGETTING break IN A switch/case STATEMENT

Erroneous Code	Corrected Code

```
char x = 'd';
...
switch (x) {
  case 'u': direction = 1;
  case 'd': direction = 2;
  case 'l': direction = 3;
  case 'r': direction = 4;
  default: direction = 0;
}
// direction = 0
```

```
char x = 'd';
...
switch (x) {
  case 'u': direction = 1; break;
  case 'd': direction = 2; break;
  case 'l': direction = 3; break;
  case 'r': direction = 4; break;
  default: direction = 0;
}
// direction = 2
```

C Code Mistake eC.9 MISSING CURLY BRACES {}

Erroneous Code	Corrected Code

```
if (ptr == NULL) // missing curly braces
  printf("Unable to open file.\n");
  exit(1);      // always executes
```

```
if (ptr == NULL) {
  printf("Unable to open file.\n");
  exit(1);
}
```

C Code Mistake eC.10 USING A FUNCTION BEFORE IT IS DECLARED

Erroneous Code	Corrected Code

```
int main(void) {
  test();
}
void test(void) {
 ...
}
```

```
void test(void) {
 ...
}
int main(void) {
  test();
}
```

C Code Mistake eC.11 DECLARING A LOCAL AND GLOBAL VARIABLE
WITH THE SAME NAME

Erroneous Code	**Corrected Code**

```
int x = 5;    // global declaration of x
int test(void) {
  int x = 3;  // local declaration of x
  ...
}
```

```
int x = 5;    // global declaration of x
int test(void) {
  int y = 3;  // local variable is y
  ...
}
```

C Code Mistake eC.12 TRYING TO INITIALIZE AN ARRAY WITH {} AFTER
DECLARATION

Erroneous Code	**Corrected Code**

```
int scores[3];
scores = {93, 81, 97}; // won't compile
```

```
int scores[3] = {93, 81, 97};
```

C Code Mistake eC.13 ASSIGNING ONE ARRAY TO ANOTHER USING =

Erroneous Code	**Corrected Code**

```
int scores[3] = {88, 79, 93};
int scores2[3];
scores2 = scores;
```

```
int scores[3] = {88, 79, 93};
int scores2[3];
for (i = 0; i < 3; i++)
  scores2[i] = scores[i];
```

C Code Mistake eC.14 FORGETTING THE SEMI-COLON AFTER A do/while
LOOP

Erroneous Code	**Corrected Code**

```
int num;
do {
  num = getNum();
} while (num < 100) // missing ;
```

```
int num;
do {
  num = getNum();
} while (num < 100);
```

C Code Mistake eC.15 USING COMMAS INSTEAD OF SEMICOLONS IN for
LOOP

Erroneous Code	**Corrected Code**

```
for (i=0, i < 200, i++)
  ...
```

```
for (i = 0; i < 200; i++)
  ...
```

C Code Mistake eC.16 INTEGER DIVISION INSTEAD OF FLOATING-POINT DIVISION

Erroneous Code

```
// integer (truncated) division occurs when
// both arguments of division are integers
float x = 9 / 4; // x = 2.0
```

Corrected Code

```
// at least one of the arguments of
// division must be a float to
// perform floating-point division
float x = 9.0 / 4; // x = 2.25
```

C Code Mistake eC.17 WRITING TO AN UNINITIALIZED POINTER

Erroneous Code

```
int *y = 77;
```

Corrected Code

```
int x, *y = &x;
*y = 77;
```

C Code Mistake eC.18 GREAT EXPECTATIONS (OR LACK THEREOF)

A common beginner error is to write an entire program (usually with little modularity) and expect it to work perfectly the first time. For nontrivial programs, writing modular code and testing the individual functions along the way are essential. Debugging becomes exponentially harder and more time-consuming with complexity.

Another common error is lacking expectations. When this happens, the programmer can only verify that the code produces a result, not that the result is correct. Testing a program with known inputs and expected results is critical in verifying functionality.

This appendix has focused on using C on a system such as a personal computer. Chapter 9 (available as a web supplement) describes how C is used to program SparkFun's RED-V RedBoard—based on a RISC-V microcontroller—that can be used in embedded systems. Microcontrollers are usually programmed in C because the language provides nearly as much low-level control of the hardware as assembly language, yet is much more succinct and faster to write.